Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Spécification et génération de SIAD dans l'atelier DB-MAIN

Volume I

Bergmann, Joël

Award date:
1996

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Spécification et génération de SIAD dans l'atelier DB-MAIN

Volume I

Joël Bergmann

Le Volume II est disponible à l'Institut d'Informatique

Mémoire présenté en vue
de l'obtention du diplôme de
Licencié et Maître en Informatique

Promoteur : J.-L. Hainaut

Année Académique 1995-1996

Résumé

A l'heure actuelle, le modèle Entité-Association (modèle EA) est un des modèles conceptuels le plus utilisé pour l'analyse et la modélisation des systèmes d'information. Ce mémoire étend la spécification du modèle EA pour permettre la modélisation d'informations dérivables : on dote le modèle EA de la possibilité d'exprimer que certaines informations d'un schéma peuvent être inférées à partir d'autres. Le modèle ainsi obtenu est appelé *modèle EA déductif.* On décrit aussi comment rendre un schéma EA déductif exécutable en transformant automatiquement les règles de définition des informations dérivables en requêtes SQL. La soumission de ces requêtes sur une base de données de production permet le calcul effectif des valeurs déduites. Le modèle EA déductif a été implémenté dans l'atelier d'ingénierie de bases de données DB-MAIN. L'intégration du modèle EA déductif dans DB-MAIN permet de spécifier et de générer des SIAD aux possibilités modestes. Grâce à ceux-ci, les gestionnaires peuvent interroger, à un niveau conceptuel, les données de production afin d'en déduire les informations dont ils ont besoin pour prendre leurs décisions.

Abstract

The Entity-Relationship Model (ER model) is currently one of the most used conceptual models for analysing and modelling information systems. This thesis extend the specification of the ER model by adding means to define derivable information : the ER model is endowed with the ability to modelize the fact that certain information in a schema can be inferred from others. The resulting model is called *Deductive ER Model.* We also describe how to make a deductive ER schema executable by translating automatically the definition rules of derivable information into SQL statements. The submission of these statements on a operational database enables the calculation of the deduced values. The Deductive ER Model has been implemented into the DB-MAIN CASE tool which is dedicated to database application engineering. The integration of the Deductive ER Model in DB-MAIN allows the specification and the generation of little DSSs. Thanks to these, the managers can query, at a conceptual level, the production data in order to deduce the information they need to make their decisions.

Avant-propos

Au moment de remettre ce travail, il m'est particulièrement agréable de pouvoir remercier toutes les personnes qui, de près ou de loin, en ont permis la réalisation.

Mes premiers remerciements s'adressent à Monsieur le professeur Jean-Luc Hainaut, promoteur du mémoire. Je lui suis particulièrement reconnaissant pour ses précieux conseils, sa constante disponibilité et l'intérêt qu'il a porté à l'élaboration de ce travail.

J'exprime également ma profonde gratitude à Monsieur Didier Rossi, *ECR Project Manager*, pour m'avoir permis de réaliser un stage des plus intéressants chez Procter & Gamble. Je le remercie pour son écoute, son dévouement, son attention et la confiance qu'il m'a témoignée dans l'attribution d'un projet considérable. J'en profite de même pour adresser mes plus vifs remerciements à toute l'équipe ECR (*Efficient Consumer Response*) pour leur accueil chaleureux, leur soutient et leur collaboration.

Ma pensée se porte également vers Vincent Englebert et toute l'équipe DB-MAIN pour leur aide et leur disponibilité tout au long du développement technique de l'application.

Je remercie aussi Monsieur Jean-Paul Leclercq pour son intervention dans certains aspects spécifiques de la réalisation du travail.

Enfin, j'aimerais remercier de tout coeur ma famille et mes amis qui n'ont jamais cessé de me soutenir et de m'aider dans l'élaboration de ce mémoire et dans l'accomplissement de mes études.

Table des matières

1. Introduction

Depuis une vingtaine d'années, le modèle Entité-Association (modèle EA) est un des modèles conceptuels le plus utilisé dans l'analyse et la modélisation des systèmes d'information. Il permet une représentation de la structure des informations indépendamment de la manière dont celles-ci sont stockées physiquement.

Ce modèle rencontre un vif succès parmi les spécialistes des systèmes d'information et des bases de données : on lui attribue de bonnes facultés à représenter les informations appartenant au réel perçu. C'est un article de P. Chen ([CHEN,76]) qui est considéré comme l'expression de référence de l'approche EA.

Dans [BODART,93], le modèle EA est caractérisé comme suit : *"Le modèle EA permet d'exprimer la sémantique des données mémorisables et/ou véhiculables à l'aide des concepts d'entité, d'association, d'attribut et du mécanisme des contraintes d'intégrité."*

Les systèmes modélisés avec le modèle EA sont classiques dans le sens où ils ne permettent pas de stocker des connaissances ou de faire des déductions sur les informations présentes dans la base de données. Le modèle EA, dans sa forme habituelle, n'autorise pas la représentation d'informations dérivables ni l'expression de règles de déduction.

Il serait pourtant intéressant de disposer, dans le modèle EA, de la possibilité d'exprimer que certaines informations ne sont pas de simples informations opérationnelles (stockées dans la base de données) mais des informations qui peuvent être déduites à partir d'autres (informations dérivées). Dans un schéma EA, on devrait pouvoir exprimer que la valeur de certains éléments est déduite des valeurs prises par d'autres éléments.

Dans ce mémoire, on se propose d'étendre la spécification du modèle EA pour permettre la modélisation du caractère dérivable de certaines informations : le modèle EA ainsi obtenu est appelé **modèle EA déductif**. On tentera également de développer et d'implémenter un système capable de rendre un schéma EA déductif exécutable en transformant automatiquement les formules de calcul des informations dérivables (règles de déduction) en requêtes SQL. La soumission de ces requêtes SQL sur une base de données réelle permet ensuite de calculer effectivement les valeurs dérivées. Cette implémentation du modèle EA déductif est réalisé dans l'atelier d'ingénierie de bases de données DB-MAIN ([DBMAIN,95]).

L'enrichissement du modèle EA par le concept d'information dérivable est d'un intérêt tout particulier dans le cadre des Systèmes d'Information d'Aide à la Décision (SIAD). Dans leurs activités décisionnelles, les gestionnaires se basent sur une multitude d'informations : faits, rumeurs, dossiers, documents non structurés (notes, rapports, textes, graphiques, etc.) ou encore données structurées en provenance des systèmes d'information opérationnels. C'est au niveau de l'exploitation, par les gestionnaires, de ce dernier type d'informations que le modèle EA déductif apporte une contribution dans le monde des SIAD.

Avec les possibilités de déduction du modèle EA déductif, les gestionnaires ont maintenant la possibilité d'exprimer, à un niveau conceptuel, leurs besoins en informations en provenance de la base de données opérationnelle. Le modèle EA déductif (et son implémentation) permet aux gestionnaires de dériver effectivement les informations dont ils

ont besoin pour prendre leur décision, pour contrôler et organiser l'entreprise en faisant des déductions sur les informations stockées dans la base de données.

L'intégration du modèle EA déductif dans l'atelier DB-MAIN dote celui-ci de la possibilité de spécifier et de générer des SIAD aux possibilités certes limitées mais néanmoins satisfaisantes pour un utilisateur aux exigences modérées.

Dans cette introduction, nous commencerons par dresser l'état de l'art en donnant un bref aperçu des travaux qui ont été réalisés dans la même approche (cfr. *1.1*). Nous définirons ensuite le concept d'élément dérivable (cfr. *1.2*) puis ceux de modèle EA standard, modèle EA de base et modèle EA déductif (cfr. *1.3*). Nous finirons par la présentation de l'organisation du mémoire (cfr. *1.4*).

1.1 Etat de l'art

Les systèmes d'information traditionnels sont orientés vers le traitement de données. Les bases de données qui y sont associées contiennent des données relatives à des entités et des associations passives : ces données correspondent à des **connaissances passives**. Toutes les **connaissances actives** sont encapsulées dans les programmes d'application qui manipulent la base de données.

Ce mode de représentation est adéquat pour un grand nombre de traitements tels que la gestion de stock, la facturation, la gestion des commandes, le payement des salaires, la comptabilité, etc. Il n'est cependant pas adapté à d'autres activités cruciales des organisations comme, par exemple, la modélisation des processus, la planification, l'allocation de ressources, la prise de décision, etc. Ces activités s'inscrivent, pour la plupart, dans le cadre des Systèmes d'Information d'Aide à la Décision. Elles requièrent des bases de données capables, non seulement, de stocker des connaissances passives mais aussi des connaissances actives sous la forme de règles.

Pour répondre à ces besoins relativement nouveaux, de nombreux efforts ont été consacrés, ces dernières années, à l'élaboration de bases de connaissances ou de bases de données déductives tant au niveau des techniques d'implémentation qu'au niveau des langages de manipulation ([ULLMAN,88]). On a notamment étendu le modèle relationnel pour permettre l'expression de connaissances sous la forme de règles ([KARAGIANNIS,87], [BLANING,87]).

Les bases de connaissances et les bases de données déductives sont aptes à représenter non seulement des connaissances passives mais aussi des connaissances actives. Elles utilisent pour ce faire des langages spécialisés tels que la logique des prédicats ou les réseaux sémantiques. On citera, par exemple, *Datalog* ([ULLMAN,88]) qui est une variante de PROLOG adaptée à la manipulation des bases de données. Ce langage repose sur le modèle relationnel : il permet de définir des règles sous la forme de relations et, ainsi, de dériver de nouvelles informations à partir des informations opérationnelles présentes dans la base de données.

L'utilisation accrue de ces nouveaux types de systèmes d'information à même de représenter la connaissance requiert, au niveau conceptuel, des outils d'analyse capables de modéliser les connaissances passives et actives de ces systèmes.

Plusieurs travaux de recherche qui visent à étendre les spécifications du modèle EA dans ce sens ont été réalisés ([LAZIMI,89], [CHEN,88], [LOUCOPOULOS,91], [SOON,91], [DECHOW,88], [RAUH,94]). Ces travaux proposent chacun une extension du modèle EA pour permettre, au niveau conceptuel, la représentation des connaissances passives et actives : informations dérivées, modélisation de processus, modèles de décision, etc.

Ces nouveaux modèles conceptuels permettent la modélisation des systèmes d'information déductifs en représentant la connaissance sous forme de règles. On intègre, au niveau conceptuel, le modèle de calcul (la connaissance) aux données. C'est cette approche que nous choisirons dans le cadre de ce mémoire. Les modèles EA développés par les auteurs que nous venons de citer permettent la modélisation d'une large variété de connaissances : informations dérivées, modèles de décision, processus, etc. Dans la cadre de ce mémoire, nous nous limiterons cependant à la modélisation des informations dérivées.

Au niveau de l'implémentation de ce type de systèmes, on citera les travaux réalisés dans [SOON,91] : il propose de rendre exécutable un schéma EA déductif (niveau conceptuel) dans une base de données déductive (niveau physique). Lors de l'implémentation d'un schéma EA dans la base de données déductive, on transforme la connaissance qui s'exprime sur le schéma EA en règles compréhensibles par la base de données déductive. Dans ce mémoire, nous implémentons un schéma EA déductif dans une base de données classique en générant un ensemble de requêtes SQL calculant les valeurs des informations dérivées : nous ne nous appuyons pas sur les bases de données déductives.

Une autre approche possible dans la représentation de la connaissance est de coupler la base de données traditionnelle à un logiciel de calcul tel que le tableur. Le tableur est l'outil de prédilection dans le domaine de l'aide à la décision : en le couplant à une base de données, il accède aux informations contenues dans celle-ci, effectue des calculs et fournit des résultats qui peuvent éventuellement être stockés dans la base de données. La modélisation des modèles de calcul étendus aux bases de données a fait l'objet de travaux dans [HAINAUT,94a] ou [HICK,91]. Dans cette approche, la base de données du système contient toutes les données tandis que le modèle de calcul renferme toute la connaissance : les données restent indépendantes mais elles sont partagées par la base de données et le tableur.

Au niveau de l'implémentation de ce type de systèmes couplant base de données et tableur, on citera, par exemple, *EMT* ([SCHIFF,88]).

1.2 Eléments dérivables

Dans un schéma EA, un **élément dérivable** est un type d'entités, un type d'associations ou un attribut dont la valeur est dérivée (calculée, déduite) à partir des valeurs prises par d'autres éléments du schéma.

Chaque élément dérivable est, par conséquent, défini par une **formule de définition** qui spécifie comment calculer sa valeur à partir de la valeur des autres éléments.

Un élément d'un schéma EA qui n'est pas dérivable est appelé **élément de base** : type d'entités de base, type d'associations de base ou attribut de base. La valeur d'un élément de base est introduite dans le système d'information et non calculée à partir d'autres éléments.

Considérons, par exemple, le modèle EA de la figure 1-1 modélisant le système d'information du département chargé de la gestion du personnel d'une société.

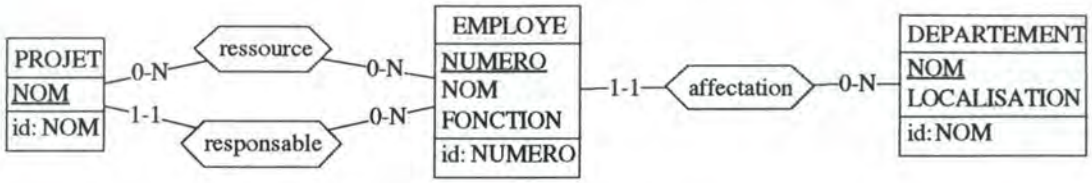


Figure 1-1 : Schéma EA

Ce schéma ne comprend que des éléments de base. On pourrait y ajouter les éléments suivants :

- type d'entités `RESPONSABLE` reprenant toutes les entités de `EMPLOYE` qui jouent au moins une fois le rôle de responsable de projet;
- attribut `NB_EMPLOYES` dans le type d'entités `DEPARTEMENT` qui représente le nombre d'employés qui travaillent dans un département;
- type d'associations `DIRECTION` reliant un employé à un département si cet employé est directeur de ce département (il faut que l'employé soit affecté à ce département et que sa fonction soit "Directeur").

Ces trois nouveaux éléments du schéma sont des éléments dérivables. On peut, en effet, calculer leur valeur à partir de celle des autres éléments du schéma. Le concept d'élément dérivable permet donc d'inférer de nouvelles données à partir de celles déjà présentes dans le système d'information.

1.3 Niveaux de modèles EA

Le modèle conceptuel que nous utilisons pour modéliser un domaine d'application est le modèle Entité-Association. Dans cette section, nous distinguons les différents modèles EA auxquels nous nous référons dans ce travail : modèle EA standard, modèle EA de base et modèle EA déductif.

Par **modèle EA standard**, nous désignons le modèle Entité-Association classique utilisé généralement par les professionnels de l'informatique. Ce modèle, relativement puissant, offre une large gamme de concepts, parfois complexes, pour la modélisation d'un domaine d'application.

Dans le cadre de ce mémoire, nous nous limitons à une variante simplifiée du modèle EA standard. Nous ferons référence à ce modèle EA simplifié en tant que **modèle EA de base**. Quoique limitée, la puissance d'expression de ce modèle est cependant suffisante pour modéliser la plupart des situations qui peuvent se présenter au concepteur de systèmes d'information. De plus, nous verrons qu'il existe une série de mécanismes permettant de transformer un schéma EA standard en un schéma EA de base.

Le modèle EA de base permet uniquement la modélisation d'éléments de base. Le **modèle EA déductif** offre, lui, la possibilité supplémentaire d'exprimer que certains éléments d'un schéma ont une valeur qui peut être calculée à partir de la valeur d'autres éléments. Le modèle EA déductif est donc une extension du modèle EA de base avec le concept d'élément dérivable.

Comme nous l'avons déjà souligné, les éléments dérivables d'un schéma EA déductif sont définis par une formule de définition ; il faut, dès lors, doter le modèle EA déductif d'un langage permettant l'expression de ces formules. [BATINI,92] et [HALPIN,95] proposent le langage naturel pour les exprimer. Nous préférons définir un langage formel.

Pour obtenir une spécification du modèle EA déductif, il suffit de reprendre la spécification du modèle EA de base et de l'enrichir par un langage permettant d'exprimer les formules de définition des éléments dérivables. Ce langage est, en effet, la seule chose qui manque au modèle EA de base pour pouvoir définir des éléments dérivables dans un schéma.

Pour conclure, on pourrait synthétiser la situation par le graphique suivant :

Figure 1-2 : Modèle EA standard/Modèle EA de base/Modèle EA déductif

Le modèle EA de base est un sous-ensemble des concepts utilisés dans le modèle EA standard. Le modèle EA déductif est une extension du modèle EA de base par le concept d'élément dérivable. Cette extension se traduit pratiquement en enrichissant le modèle EA de base d'un langage permettant l'expression des formules de définition.

1.4 Organisation du mémoire

Le mémoire est constitué de deux volumes. Le premier est organisé comme suit :

Le chapitre 2 définit le modèle EA standard. Le chapitre 3 spécifie le modèle EA de base et présente un certain nombre de mécanismes de transformation de schéma qui permettent de passer d'un schéma EA standard à un schéma EA de base.

Le chapitre 4 présente le modèle EA déductif en définissant, d'abord de manière intuitive puis de manière plus formelle, le langage d'expression des formules de définition. Il indique également les limitations du langage et propose des moyens pour y remédier.

Une fois le modèle EA déductif défini, le chapitre 5 étudie la manière d'exploiter celui-ci. On envisage les différentes méthodes pour rendre un schéma EA déductif exécutable. On choisira finalement un mode d'exploitation particulier et on basera la suite du travail sur celui-ci.

Le chapitre 6 définit deux concepts importants relatifs au modèle EA déductif qui seront particulièrement utiles pour la suite du travail : celui de cardinalités d'une expression de désignation d'un attribut et celui de graphe de dépendance d'un schéma EA déductif. Le chapitre 7 propose un ensemble de règles de cohérence que tout schéma EA déductif doit respecter.

Le chapitre 8 approfondit un mode d'exploitation particulier du modèle EA déductif : il spécifie comment implémenter un système qui calcule automatiquement, dans une base de données réelle, les valeurs des informations dérivables en transformant les formules de définition de celles-ci en requêtes SQL. Le chapitre 9 explique brièvement comment nous avons implémenté pratiquement le processus développé au chapitre 8 dans l'atelier DB-MAIN d'ingénierie de bases de données.

Le chapitre 10 propose finalement quelques pistes pour prolonger le travail commencé dans ce mémoire.

Le second volume, disponible sur demande, contient les spécifications techniques détaillées des programmes implémentés ainsi que le code source de ceux-ci.

2. *Modèle EA standard*

Ce chapitre définit, sur base d'exemples, les différents concepts du modèle EA standard. Il existe de nombreuses variantes du modèle EA notamment au niveau de la représentation graphique. Nous adoptons une approche inspirée de [HAINAUT,94a], [BODART,93] ou encore [DBMAIN,95].

2.1 Types d'entités

Le domaine d'application est perçu comme étant constitué d'entités concrètes ou abstraites. Dans le contexte d'une société de vente par correspondance, on peut cerner un domaine d'application dans lequel on repère des clients, des commandes, des produits et des fournisseurs. On considère que chacun d'eux est une **entité** du domaine et que chaque entité appartient à une classe ou **type d'entités**. On définit naturellement les quatre types d'entités `CLIENT`, `COMMANDE`, `PRODUIT` et `FOURNISSEUR`.

Ces types d'entités sont représentés graphiquement comme suit :

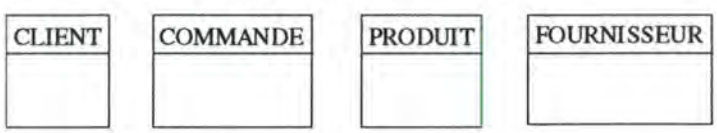


Figure 2-1 : Types d'entités du domaine d'application

2.2 Attributs

Chaque entité est caractérisée par un certain nombre d'**attributs**. Ainsi, chaque produit est caractérisé par un numéro, une description, un prix de vente, un prix d'achat et une quantité en stock. Un attribut peut être considéré comme une propriété d'une entité. Un attribut possède un **type** : numérique, caractère ou booléen. La figure suivante montre comment représenter graphiquement les attributs des types d'entités retenus jusqu'ici :

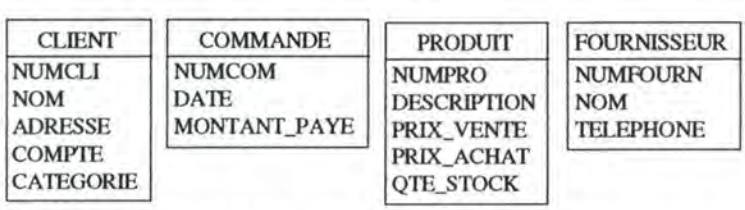


Figure 2-2 : Types d'entités avec attribut

2.2.1 Attribut facultatif

Il est possible que, pour une entité donnée, la valeur d'un attribut ne soit pas connue ou que cette valeur n'ait pas de sens pour cette entité. Si on admet qu'un attribut peut ne pas avoir de valeur pour certaines entités, on le déclare comme **attribut facultatif** et comme **attribut obligatoire** dans le cas contraire. Graphiquement, un attribut facultatif est indiqué par la notation `[0-1]` qui signifie que, pour une entité donnée, cet attribut peut prendre 0 ou 1 valeur. Dans notre exemple, supposons que le numéro de téléphone d'un fournisseur ne soit pas nécessairement connu : l'attribut `TELEPHONE` de `FOURNISSEUR` est facultatif. Le type d'entités `FOURNISSEUR` se présente maintenant comme illustré à la figure 2-3.

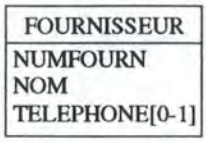


Figure 2-3 : Attributs obligatoires et facultatifs

2.2.2 Attribut multivalué

Un attribut peut prendre plus d'une valeur pour une entité donnée. On dit alors que cet attribut est **multivalué**. Un tel attribut est caractérisé par un couple de valeurs `[i-j]` représentant le nombre minimum (`i`) et le nombre maximum (`j`) de valeurs que cet attribut peut prendre pour une entité donnée. Toute valeur non négative est admise pour `i` et `j` à condition que $i \leq j$ et que $j \geq 1$.

Si on désire, par exemple, connaître tous les prénoms d'un client et qu'on en retient au maximum 10 (et au minimum 1), le type d'entités `CLIENT` devient :

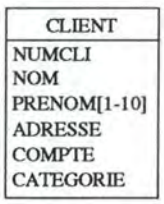


Figure 2-4 : Attribut multivalué

Remarque : Un attribut facultatif n'est, en fait, qu'un cas particulier d'attribut multivalué où `i` vaut 0 et `j` vaut 1.

2.2.3 Attribut décomposable

Lorsque la valeur d'un attribut peut être désagrégée en valeurs significatives plus petites, on dit que l'attribut est **décomposable**. Dans notre exemple, on peut voir l'attribut `ADRESSE` de `CLIENT` comme étant composé d'une valeur `RUE` + une valeur `LOCALITE` + une valeur `CODE_POSTAL`. Graphiquement, les attributs décomposables se représentent ainsi :

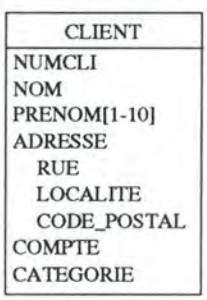


Figure 2-5 : Attribut décomposable

Un attribut qui n'est pas décomposable est appelé **attribut atomique**.

2.3 Types d'associations

Souvent, les différentes entités du domaine d'application sont liées entre elles par des **associations**. Dans notre exemple, une commande est passée par un client : il y a une association entre une commande et le client qui l'a passée. Chaque association appartient à une classe ou **type d'associations**. Ainsi, on dira qu'il existe un type d'associations (appelé ici `passe`) entre les types d'entités `CLIENT` et `COMMANDE`. Les types d'associations sont représentés graphiquement comme suit :

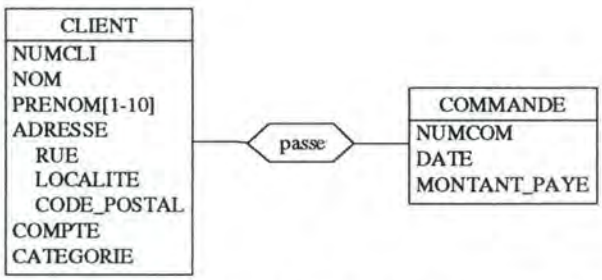


Figure 2-6 : Types d'entités, attributs et types d'associations

Un type d'associations peut relier un nombre quelconque de types d'entités. Le nombre de types d'entités participant à un type d'associations est appelé le **degré du type d'associations**. Considérons la partie suivante du schéma :

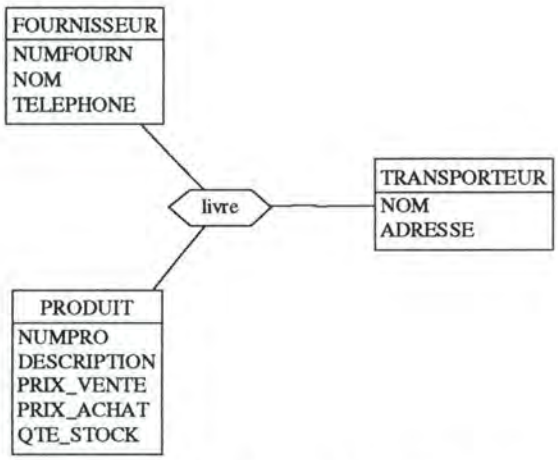


Figure 2-7 : Type d'associations de degré 3

Cette partie de schéma illustre le fait qu'un produit est livré par un fournisseur et qu'il est acheminé du fournisseur vers l'entreprise par un transporteur. Une association `livre` associe un produit à un fournisseur et à un transporteur.

Chaque type d'entités `TE` participant à un type d'associations `ta` joue un **rôle** dans `ta`. Sans que cela soit obligatoire, on peut nommer les rôles des types d'entités au sein des types d'associations. Par exemple, dans la figure 2-6, le rôle joué par `COMMANDE` dans `passe` peut être nommé `est passée par` tandis que celui joué par `CLIENT` peut être nommé `a passé`. Le schéma peut alors se lire : un client a passé des commandes et une commande est passée par un client.

2.3.1 Cardinalités d'un type d'associations

Les **cardinalités d'un type d'associations** ta entre des types d'entités TE_1 ... TE_n précisent à combien d'associations ta chaque entité de TE_1 ... TE_n peut participer. On indique ces cardinalités en précisant, pour chaque type d'entités TE_i participant à ta, le nombre minimum et le nombre maximum d'associations ta auxquelles une entité de TE_i peut participer. Ainsi, chaque type d'entités participant dans un type d'associations est caractérisé par un couple de valeurs i-j appelées **cardinalité minimale** (i) et **cardinalité maximale** (j). Toute valeur non négative est admise pour i et j à condition que $i \leq j$ et que $j \geq 1$. Si la cardinalité maximale est indéfinie (une entité de TE_i peut participer à un nombre quelconque d'associations ta), on précise une cardinalité maximale de N.

Dans notre exemple, une entité CLIENT peut passer plusieurs commandes (un nombre indéfini) ou alors n'en passer aucune (on accepte les clients "potentiels"). Les cardinalités du type d'entités CLIENT dans le type d'associations passe sont indiquées par 0-N : un client peut passer de 0 (cardinalité minimale) à N (cardinalité maximale) commandes. En outre, une commande est toujours passée par un et un seul client : les cardinalités de COMMANDE dans passe sont 1-1 (une commande est passée par 1 client au minimum et 1 client au maximum).

Si, dans un type d'associations ta reliant des types d'entités $TE_1 \ldots TE_n$, TE_i ($1 \leq i \leq n$) a une cardinalité minimale de 0, on dit que le type d'associations ta est **facultatif** pour TE_i.

En indiquant graphiquement les cardinalités, la figure 2-6 devient :

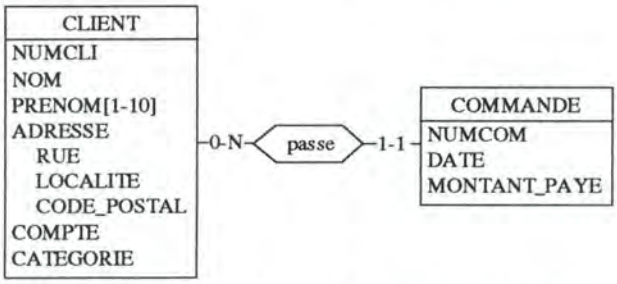


Figure 2-8 : Cardinalités des types d'entités participant à un type d'associations

C'est sur base des cardinalités d'un type d'associations qu'on peut déterminer sa **classe fonctionnelle**[1] : un-à-un, un-à-plusieurs ou plusieurs-à-plusieurs. La classe fonctionnelle d'un type d'associations ta entre deux types d'entités A et B décrit le nombre maximum d'entités B pour chaque entité A et inversement :

- si la cardinalité maximale d'un des deux types d'entités est 1 et que l'autre est supérieure à 1, on dit que ta est un type d'associations un-à-plusieurs;
- si la cardinalité maximale des deux types d'entités est supérieure à 1, on dit que ta est un type d'associations plusieurs-à-plusieurs;
- si la cardinalité maximale des deux types d'entités est 1, on dit que ta est un type d'associations un-à-un;

[1] On définit le concept de classe fonctionnelle uniquement pour les types d'associations de degré 2.

2.3.2 Types d'associations récursifs

Il est possible d'établir un type d'associations entre un type d'entités et lui-même : on définit ainsi un **type d'associations récursif**. Par exemple, on peut envisager qu'un produit soit constitué de plusieurs autres produits et que, inversement, un produit entre dans la constitution de plusieurs autres. Dans le cas d'un type d'associations récursif, il faut nommer le rôle joué par chaque type d'entités dans le type d'associations.

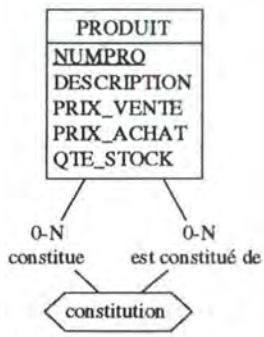


Figure 2-9 : Type d'associations récursif

2.3.3 Attributs d'un type d'associations

Comme les types d'entités, les types d'associations peuvent être caractérisés par des attributs. On peut, par exemple, imaginer un type d'associations `lignecom` entre `COMMANDE` et `PRODUIT` associant, à une commande, tous les produits commandés dans celle-ci. Une association `lignecom` reliant une commande `c` à un produit `p` est caractérisée par l'attribut `QCOM` représentant la quantité du produit `p` commandée dans `c`. Graphiquement, on représente cette situation comme ceci :

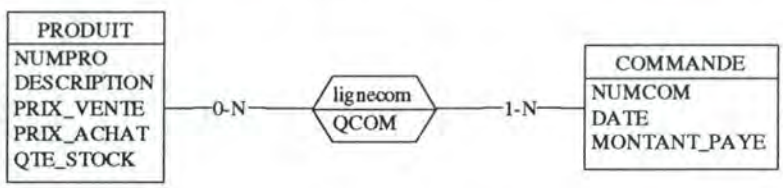


Figure 2-10 : Type d'associations avec attribut

2.4 Identifiants

Chaque type d'entités doit posséder un **identifiant** qui permet de repérer univoquement chaque entité de ce type. Par exemple, les entités du type `CLIENT` sont identifiées par leur numéro de client (`NUMCLI`) unique. Cela signifie qu'à tout instant, il n'existe pas deux entités `CLIENT` qui possèdent la même valeur de `NUMCLI`. On dit que `CLIENT` est identifié par `NUMCLI`.

Le cas du type d'entités `COMMANDE` est plus complexe. On admet, en effet, que les commandes reçoivent un numéro qui les identifie parmi celles du client qui les a passées. Il n'existe donc pas deux commandes portant le même numéro et passées par le même client. On peut donc dire que le type d'entités `COMMANDE` est identifié par l'attribut `NUMERO` et par le type d'entités `CLIENT`.

En toute généralité, un identifiant d'un type d'entités TE peut avoir l'une des compositions suivantes :

1. un attribut de TE;
2. plusieurs attributs de TE;
3. un ou plusieurs attributs de TE + un ou plusieurs types d'entités TE_i associés à TE via le ou les types d'associations ta_i;
4. plusieurs types d'entités TE_i associés via ta_i à TE.

Quand un type d'entités TE_i intervient dans l'identifiant de TE (cas 3 et 4 ci-dessus), on dit que ce dernier est un **identifiant hybride**.

Les constituants de l'identifiant d'un type d'entités TE sont indiqués graphiquement sous le rectangle représentant TE avec la notation "id:...". Lorsque l'identifiant est constitué, en tout ou en partie, d'un type d'entités TE_i relié à TE via un type d'associations ta_i, on indique ce constituant de l'identifiant par la notation $ta_i.TE_i$. Si l'identifiant de TE est constitué uniquement d'attributs de TE, alors ces attributs sont soulignés.

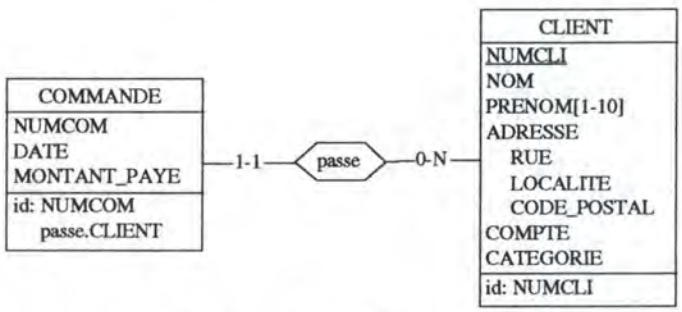


Figure 2-11 : Identifiant d'un type d'entités

Tout attribut participant à l'identifiant d'un type d'entités TE doit être défini comme obligatoire. De plus, si l'identifiant de TE est constitué, en tout ou en partie, d'un type d'entités TE_i relié à TE via un type d'associations ta_i, il faut obligatoirement que les cardinalités de TE dans ta_i soient 1-1.

Remarque : Tout type d'associations ta est implicitement identifié par tous les types d'entités TE_i qui participent à ta.

2.5 Sous-types

Supposons que notre société de vente par correspondance possède deux types de clients : des particuliers et des entreprises. Les particuliers et les entreprises sont des catégories distinctes de clients : on définit deux nouveaux types d'entités PARTICULIER et ENTREPRISE qui sont des **sous-types** de CLIENT. CLIENT est appelé le **sur-type**.

Chaque sous-type hérite de toutes les propriétés du sur-type (attributs, identifiant, types d'associations) et peut posséder des propriétés qui lui sont propres. Dans notre exemple, un particulier et une entreprise sont tous deux caractérisés par les attributs NUMCLI, NOM, ADRESSE, COMPTE et CATEGORIE, ils sont identifiés par l'attribut NUMCLI et ils sont associés, via le type d'associations passe, aux commandes qu'ils ont passées.

En plus des propriétés communes héritées du sur-type `CLIENT`, chaque sous-type est caractérisé par des attributs propres :

- `PARTICULIER` possède les attributs `PRENOM` et `ETAT_CIVIL`;
- `ENTREPRISE` possède l'attribut `REG_COMMERCE` qui représente le numéro du registre de commerce de l'entreprise.

Chaque entité du sur-type doit obligatoirement appartenir à un et un seul de ses sous-types. La relation de sous-typage se représente graphiquement par un triangle relié au sur-type par une "patte grasse" et à chaque sous-type par une "patte maigre" (cfr. figure 2-12).

Dans sa forme définitive, notre exemple se présente graphiquement comme suit :

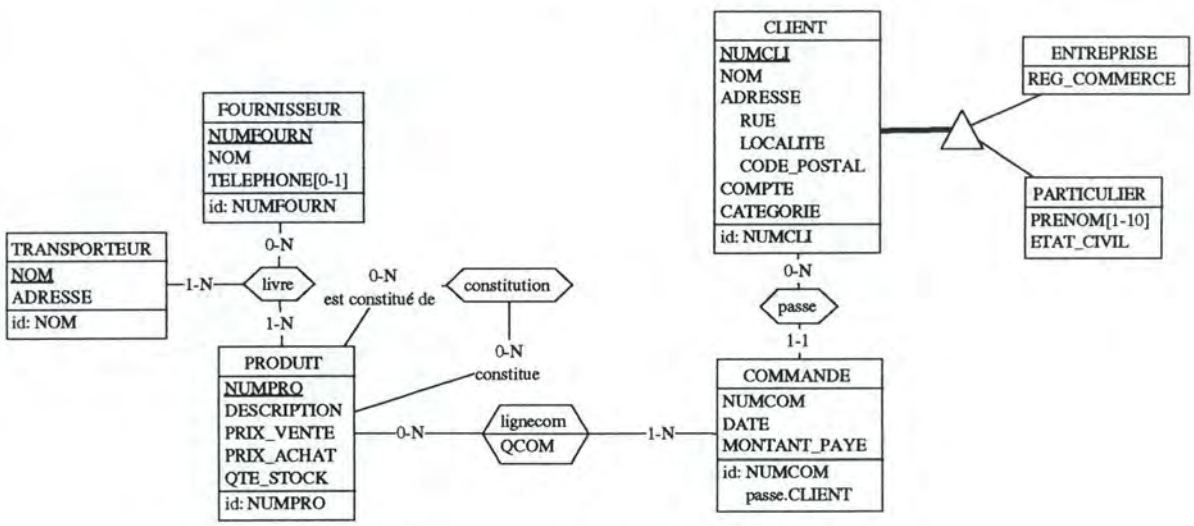


Figure 2-12 : Version finale du schéma EA standard

3. *Modèle EA de base*

Comme nous l'avons stipulé dans l'introduction, le modèle EA de base est un sous-ensemble des formalismes utilisés dans le modèle EA standard. Dans ce chapitre, nous commençons par définir le modèle EA de base en précisant ses limitations par rapport au modèle EA standard. Nous étudions ensuite une série de mécanismes qui permettent de transformer un schéma EA standard en un schéma EA de base.

La définition du modèle EA de base s'appuie sur le modèle simplifié présenté dans [HAINAUT,94a].

3.1 Définition du modèle EA de base

Le modèle EA de base repose sur la définition du modèle EA standard. Les constructions suivantes du modèle EA standard sont proscrites dans le modèle EA de base :

- un type d'entités ne peut pas être déclaré sous-type d'un autre;
- un type d'associations peut relier au plus deux types d'entités : on admet uniquement les types d'associations binaires (de degré 2);
- un type d'associations ne peut pas avoir d'attributs;
- les seules valeurs de cardinalités admises pour un type d'entités participant à un type d'associations sont `1-1`, `0-1`, `1-N` et `0-N`;
- on n'admet ni les attributs décomposables, ni les attributs multivalués. Les attributs facultatifs sont acceptés.

3.2 Transformations de schémas

On explique maintenant comment transformer les constructions du modèle EA standard proscrites dans le modèle EA de base en constructions équivalentes. Ces transformations de schémas sont extraites de [HAINAUT,94b] et [DBMAIN,95].

La plupart des transformations sont à **sémantique constante** ce qui signifie que les transformations modifient la syntaxe d'un schéma mais pas sa sémantique : les deux schémas (avant et après transformation) décrivent strictement le même domaine d'application. Lorsqu'une transformation n'est pas à sémantique constante, nous l'indiquerons expressément.

3.2.1 Transformation d'une structure de spécialisation

Quand on est en présence d'un sous-type, il suffit de transformer celui-ci en un type d'entités classique et de le relier au sur-type par un type d'associations.

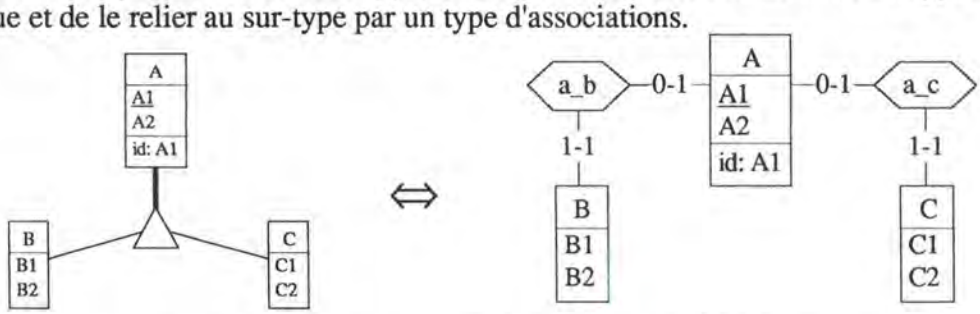


Figure 3-1 : Transformation d'une structure de spécialisation

B et C sont identifiés implicitement par A : pour une entité de A il existe, en effet, au plus une entité de B ou de C qui lui est associée.

3.2.2 Transformation des cardinalités d'un type d'associations

Les seules cardinalités admises pour un type d'associations sont 1-1, 0-1, 1-N et 0-N. Pour transformer des cardinalités quelconques i-j d'un type d'entités E dans un type d'associations ta, on suivra les règles suivantes :

- si i > 1 alors la cardinalité minimale de E dans ta vaut 1, sinon elle vaut i;
- si j > 1 alors la cardinalité maximale de E dans ta vaut N, sinon elle vaut j.

La transformation des cardinalités ne conserve pas la sémantique du schéma de départ.

3.2.3 Transformation d'un type d'associations complexe

Quand on est en présence d'un type d'associations complexe ta (type d'associations de degré supérieur à 2 ou type d'associations avec attributs), on transforme ce type d'associations en un type d'entités et on relie ce nouveau type d'entités à chaque type d'entités participant à ta par des types d'associations binaires. L'identifiant du type d'entités ainsi créé est constitué des types d'entités participant dans ta.

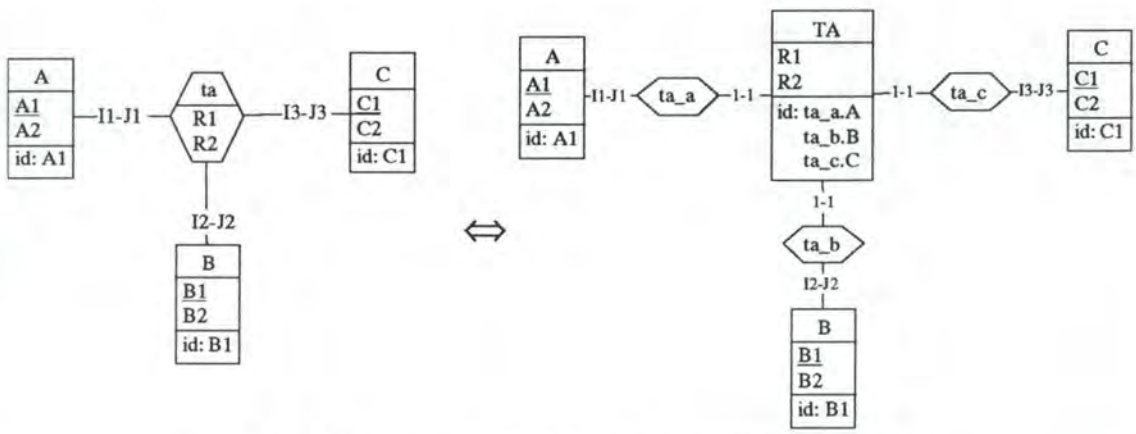


Figure 3-2 : Transformation d'un type d'associations complexe

On applique les règles énoncées en *3.2.2* pour transformer les cardinalités quelconques i-j en cardinalités admises par le modèle EA de base.

3.2.4 Transformation d'un attribut multivalué

Pour transformer un attribut multivalué A caractérisé par le couple de valeurs [i-j], on procède par représentation des valeurs : on crée un nouveau type d'entités TA possédant un attribut A monovalué. On relie ensuite TA à E par un type d'associations ta_e. Les cardinalités de TA dans ta_e sont 1-1 et celles de E dans ta_e sont i-j.

L'identifiant de TA est constitué de l'attribut A et du type d'entités E.

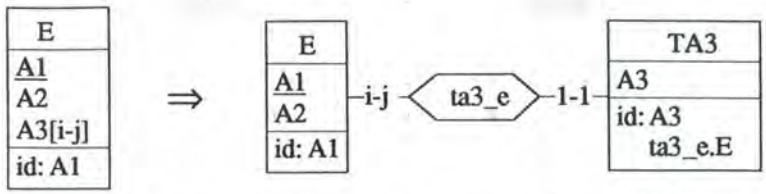


Figure 3-3 : Transformation d'un attribut multivalué

On applique les règles énoncées en *3.2.2* pour transformer `i-j` en cardinalités admises par le modèle EA de base.

3.2.5 Transformation d'un attribut décomposable

Pour transformer un attribut décomposable, on procède par désagrégation : on ramène les attributs décomposés au niveau le plus haut en les renommant éventuellement.

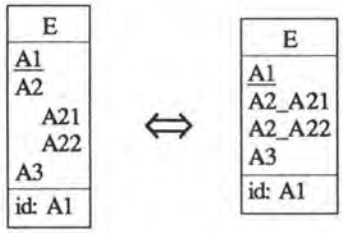


Figure 3-4 : Transformation d'un attribut décomposable

3.3 Exemple

En appliquant les transformations que nous venons de présenter sur le schéma EA standard de la figure 2-12, nous obtenons le schéma EA de base suivant :

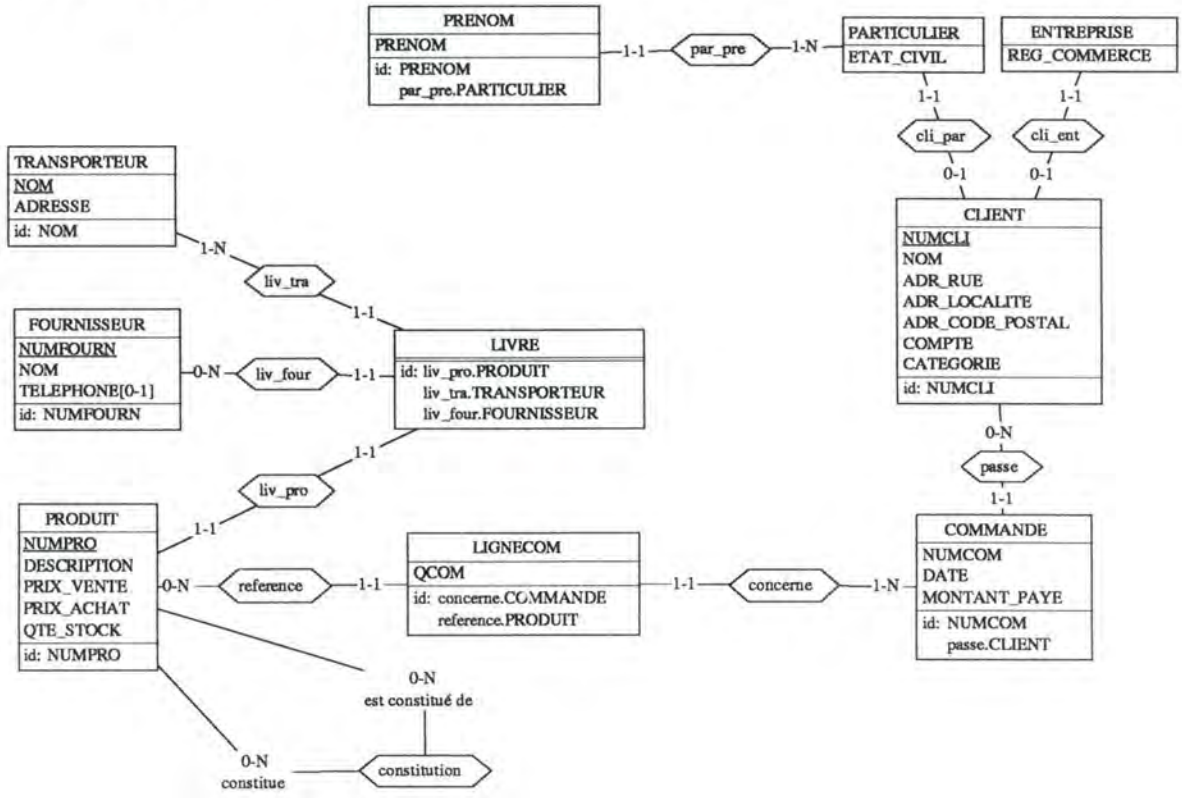


Figure 3-5 : Schéma EA de base correspondant au schéma EA standard de la figure 2-12

4. Modèle EA déductif

4.1 Introduction

Comme nous l'avons déjà défini, le modèle EA déductif constitue une extension du modèle EA de base par le concept d'élément dérivable. Pratiquement, cette extension se traduit en enrichissant le modèle EA de base par un langage d'expression des formules de définition des éléments dérivables. On obtient ainsi la spécification du modèle EA déductif.

Le concept général d'élément dérivable sera limité, dans le cadre de ce mémoire, à celui d'attribut dérivable. Dans un schéma EA déductif, il existe donc deux types d'attributs :

- les attributs de base : attribut dont la valeur est introduite par l'utilisateur dans le système d'information;
- les attributs dérivables : attribut dont la valeur est déduite de la valeur d'autres attributs.

Il n'existe cependant pas de différence fondamentale entre ces deux types d'attributs : la seule chose qui les différencie réside dans la manière d'obtenir leur valeur (par une simple consultation des données pour les attributs de base et par un calcul pour les attributs dérivables). On distingue un attribut dérivable d'un attribut de base suivant que l'attribut possède ou non une formule de définition.

Les formules de définition des attributs dérivables n'apparaissent pas, graphiquement, dans un schéma EA déductif. Un tel schéma doit, par conséquent, toujours être accompagné d'un document textuel qui donne la formule de définition de chaque attribut dérivable.

L'objectif de ce chapitre est de spécifier le langage d'expression des formules de définition des attributs dérivables. On donne d'abord une introduction intuitive au langage et ensuite une définition syntaxique et sémantique formelle. En plus des exemples introduits dans ce chapitre, l'annexe I propose un schéma EA déductif accompagné de toutes les formules de définition des attributs dérivables : ce schéma contient un grand nombre d'attributs dérivables et illustre ainsi la majorité des constructions du langage. De plus, l'annexe II présente l'étude d'un cas pratique utilisant le modèle EA déductif : il contient, lui aussi, de nombreux exemples de formules de définition.

4.2 Présentation générale du langage

Nous appuierons l'approche intuitive au langage par des exemples basés sur le schéma EA déductif de la figure 4-1. Ce schéma correspond, en fait, à une version simplifiée du schéma EA de base de la figure 3-5, complété par un certain nombre d'attributs dérivables. Les définitions des attributs dérivables seront introduites, petit à petit, dans les paragraphes suivants. Pour permettre au lecteur de différencier attribut de base et attribut dérivable, les attributs dérivables sont indiqués en italique dans le schéma.

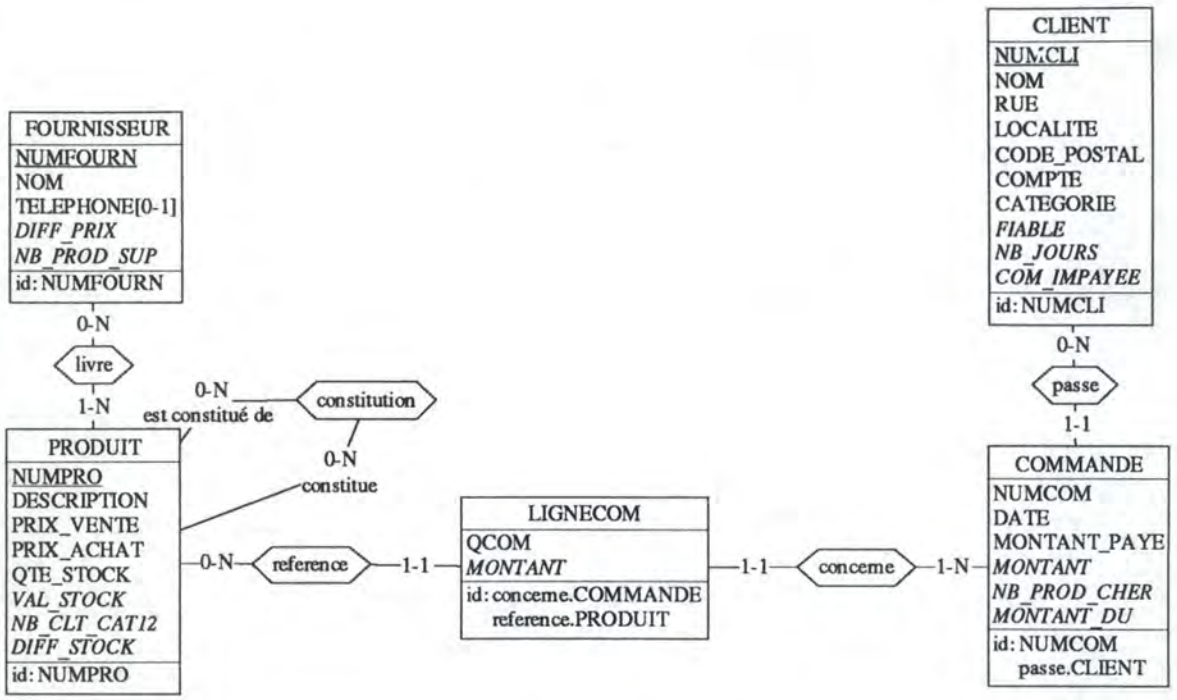


Figure 4-1 : Schéma EA déductif

La formule de définition d'un attribut dérivable est constituée d'une expression : expression simple, expression arithmétique ou expression logique. Par convention, si `Expr` est l'expression qui permet de calculer la valeur d'un attribut dérivable `A`, la formule de définition de `A` se présente ainsi [2]: `DEF=Expr`

Avant d'entamer une description complète du langage, donnons quelques exemples intuitifs de formules de définition :

- l'attribut `VAL_STOCK` de `PRODUIT` représente la valeur du stock pour un produit. Cette valeur est obtenue en multipliant la quantité du stock pour ce produit (attribut `QTE_STOCK`) par son prix d'achat (attribut `PRIX_ACHAT`). La formule de définition de `VAL_STOCK` s'écrit :

```
DEF= $.PRIX_ACHAT * $.QTE_STOCK
```

- l'attribut `MONTANT` de `LIGNECOM` représente le montant d'une ligne de commande. Ce montant est obtenu en multipliant le prix de vente du produit désigné dans la ligne (attribut `PRIX_VENTE` de `PRODUIT`) par la quantité commandée de ce produit (attribut `QCOM` de `LIGNECOM`). La formule de définition de `QCOM` s'écrit :

```
DEF= $.QCOM * PRODUIT(reference:$).PRIX_VENTE
```

[2] En pratique, `DEF` s'écrit `#DEF` et la formule de définition se termine par le symbole # : `#DEF= Expr#`. Nous ne mentionnerons cependant pas les symboles # de début et de fin pour ne pas alourdir inutilement l'exposé par des considérations purement syntaxiques.

4.2.1 Désignation d'un ensemble d'entités

On explique ici comment désigner un ensemble d'entités dans un schéma EA déductif. Il existe trois façons de désigner un tel ensemble : soit on désigne toutes les entités d'un type d'entités, soit on désigne un sous-ensemble des entités d'un type, soit on désigne l'entité courante.

Pour désigner l'ensemble de toutes les entités d'un type d'entités, on utilise le nom de ce type.

Exemple : `CLIENT` désigne l'ensemble de toutes les entités du type `CLIENT`.

Si on veut désigner une partie seulement des entités d'un type d'entités `TE`, on soumet les entités de `TE` à une **condition de sélection** qu'on place après le nom du type d'entités (après `TE`). Une condition de sélection peut porter sur la valeur prise par un attribut d'une entité de `TE` : dans ce cas, on ne retient que les entités de `TE` pour lesquelles la valeur de l'attribut en question vérifie la condition.

Exemple : `PRODUIT(:PRIX_VENTE < 1000)` désigne l'ensemble des entités `PRODUIT` dont le prix de vente est inférieur à 1000.

Une condition de sélection peut aussi porter sur les associations qui relient les entités de `TE` à un ensemble d'entités `E'` via un type d'associations `ta` : dans ce cas, on ne retient que les entités de `TE` reliées, via `ta`, à une entité au moins de l'ensemble `E'`.

Exemple : `COMMANDE(passe:CLIENT(:NUMCLI=123))` désigne l'ensemble des entités de `COMMANDE` reliées via `passe` à l'ensemble des entités de `CLIENT` dont la valeur de `NUMCLI` vaut `123`. En d'autres mots, cette expression désigne l'ensemble des commandes passées par le client de numéro `123`.

Les conditions de sélection peuvent être combinées avec les opérateurs `and`, `or` et `not`.

Exemple :
```
COMMANDE(passe:CLIENT(:COMPTE>1000 or :LOCALITE="LIEGE")
            and not :DATE > 311295
            and concerne:LIGNECOM(reference:PRODUIT
                        (:DESCRIPTION="ROBOT DE CUISINE")))
```
désigne l'ensemble des commandes qui font référence à un produit dont la description est "Robot de cuisine" et qui ont été passées avant 1996 par des clients habitant Liège ou des clients dont le montant du compte est supérieur à 1000.

Introduisons maintenant le concept d'**entité courante**. La formule de définition d'un attribut dérivable d'un type d'entités spécifie la manière dont la valeur de cet attribut doit être calculée pour toutes les entités de ce type. Il s'agit d'une formule générale qui s'applique à toutes les entités d'un type. La formule doit donc être personnalisable à chaque entité particulière. Il faudrait donc disposer d'un mécanisme permettant d'adapter une formule de définition à chaque entité particulière d'un type. Pour ce faire, on dispose du concept d'entité courante.

L'entité courante est l'entité pour laquelle on est en train de calculer la valeur de l'attribut dérivable. On y fait référence en utilisant le symbole '`$`'. Si on spécifie la formule de

définition d'un attribut dérivable a du type d'entités TE, $ symbolise "l'entité de TE pour laquelle on est en train de calculer la valeur de a".

Exemple : Dans la définition de l'attribut MONTANT de LIGNECOM, pour désigner l'entité de PRODUIT reliée, via reference, à l'entité courante de LIGNECOM, on utilise l'expression : PRODUIT(reference:$). Si $\{l_1, l_2, l_3\}$ est l'ensemble des entités de LIGNECOM, l'expression s'interprète comme suit :

- PRODUIT(reference:l_1) quand on calcule la valeur de MONTANT pour l_1
- PRODUIT(reference:l_2) quand on calcule la valeur de MONTANT pour l_2
- PRODUIT(reference:l_3) quand on calcule la valeur de MONTANT pour l_3

Le cas des types d'associations récursifs pose problème pour les conditions de sélection portant sur les associations : il est, en effet, nécessaire de préciser dans quel sens parcourir le type d'associations. Pour résoudre ce problème, on peut remplacer le nom d'un type d'entités, dans une condition de sélection portant sur les associations, par le nom du rôle joué par ce type d'entités dans le type d'associations.

Exemple : Si on spécifie la formule de définition d'un attribut dérivable de PRODUIT, l'expression est_constitué_de(constitution:$) représente toutes les entités de PRODUIT qui jouent le rôle est_constitué_de dans une association les reliant à l'entité courante. En d'autres mots, cette expression représente tous les produits qui sont constitués du produit courant. Inversement, l'expression constitue(constitution:$) représente tous les produits qui constituent le produit courant.

4.2.2 Désignation d'un attribut

Pour désigner les valeurs prises par un ensemble d'entités E pour un attribut a, on fait suivre l'expression qui désigne cet ensemble d'entités E par le nom de l'attribut a : E.a

Exemple :

- PRODUIT.PRIX_VENTE désigne l'ensemble des valeurs de l'attribut PRIX_VENTE prises par toutes les entités de PRODUIT.
- PRODUIT(:PRIX_VENTE >1000).NUMPRO désigne l'ensemble des numéros de produit dont le prix de vente est supérieur à 1000.

Nous en savons maintenant assez pour construire des formules de définition complètes :

- Si l'attribut VAL_STOCK de PRODUIT représente la valeur du stock pour un produit dépréciée de 10 %, sa formule de définition s'écrit :

```
DEF= ($.PRIX_ACHAT * (9/10)) * $.QTE_STOCK
```

- Si l'attribut FIABLE de CLIENT représente le fait qu'un client est fiable (TRUE) ou non (FALSE) et en considérant qu'un client est fiable si le montant de son compte est supérieur à 10000, alors la formule de définition de FIABLE s'écrit :

```
DEF= $.COMPTE > 10000
```

- Si l'attribut MONTANT de LIGNECOM représente le montant total d'une ligne de commande, alors sa formule de définition s'écrit :

```
DEF= $.QCOM * PRODUIT(reference:$).PRIX_VENTE
```

4.2.3 Fonctions agrégatives

Le langage d'expression des formules de définition dispose d'un certain nombre de fonctions qui permettent d'effectuer des calculs sur des éléments d'un schéma EA déductif.

Fonction `Somme`

La fonction `Somme` permet d'additionner un certain nombre de valeurs. Si l'attribut `MONTANT` de `COMMANDE` représente le montant total d'une commande, la formule de définition de `MONTANT` s'écrit :

```
DEF= Somme(I=LIGNECOM(concerne:$) ; I.QCOM * PRODUIT(reference:I))
```

Cette expression exige quelques explications. Comme on peut le constater, l'utilisation de la fonction `Somme` requiert deux arguments séparés par le caractère ';' :

- Le premier argument désigne un ensemble d'entités : `LIGNECOM(concerne:$)`.
  La fonction `Somme` calcule une certaine valeur pour chaque entité de cet ensemble et finit par renvoyer la somme de ces valeurs.
- Le deuxième argument précise comment calculer la valeur associée à chaque entité `I` de l'ensemble défini par le premier argument : `I.QCOM * PRODUIT(reference:I)`.
  `I` peut être comparé à un indice: il permet de désigner successivement chaque entité de l'ensemble `LIGNECOM(concerne:$)` et de calculer pour chacune de ces entités la valeur associée.

En résumé, on pourrait écrire :

```
Somme(I=LIGNECOM(concerne:$) ; I.QCOM * PRODUIT(reference:I))
```

$$\equiv$$

$$\sum_{\texttt{I=LIGNECOM(concerne:\$)}} \texttt{I.QCOM * PRODUIT(reference:I)}$$

Fonctions `Min` et `Max`

Les fonctions `Min` et `Max` permettent de calculer respectivement la valeur minimale ou maximale d'un ensemble de valeurs. Elles se présentent de la même manière que la fonction `Somme` : le premier argument désigne un ensemble d'entités et le second précise comment établir la valeur correspondant à chaque entité de cet ensemble. Les fonctions `Min` et `Max` établissent la valeur associée à chaque entité de l'ensemble et renvoient respectivement le minimum ou le maximum de ces valeurs.

Si l'attribut dérivable `DIFF_PRIX` de `FOURNISSEUR` désigne la différence entre le prix de vente maximal des produits livrés par ce fournisseur et le prix de vente minimal de ces produits, alors la formule de définition de `DIFF_PRIX` s'écrit :

```
DEF=  Max(I=PRODUIT(livre:$);I.PRIX_VENTE) -
      Min(I=PRODUIT(livre:$);I.PRIX_VENTE)
```

Fonction `Moyenne`

La fonction `Moyenne` permet de calculer la valeur moyenne d'un ensemble de valeurs. Elle se présente de la même manière que les fonctions précédentes. Si l'attribut dérivable `NB_PROD_CHER` de `COMMANDE` désigne la quantité de produits chers contenus dans une commande (un produit est cher si son prix est supérieur à la moyenne des prix des produits), alors la formule de définition de `NB_PROD_CHER` s'écrit :

```
DEF= Somme(I=LIGNECOM(          concerne:$
                        and     reference:PRODUIT(:PRIX_VENTE >
                                    Moyenne(I=PRODUIT;I.PRIX_VENTE)));
          I.QCOM)
```

Fonction `NombreVal`

La fonction `NombreVal` permet de compter le nombre de valeurs distinctes comprises dans un ensemble de valeurs. Elle se présente de la même manière que les fonctions précédentes. Si `NB_JOURS` de `CLIENT` représente le nombre de jours distincts de 1995 où un client a passé une commande, alors la formule de définition de `NB_JOURS` s'écrit :

```
DEF=NombreVal(I=COMMANDE(passe:$    and :DATE>=950101
                                    and :DATE<=951231);I.DATE)
```

Fonction `Nombre`

La fonction `Nombre` permet de compter le nombre d'entités distinctes présentes dans un ensemble d'entités. Contrairement aux fonctions précédentes, `Nombre` possède un seul argument qui désigne un ensemble d'entités. Si `NB_CLT_CAT12` de `PRODUIT` représente le nombre de clients distincts de catégorie 1 ou 2 qui ont acheté le produit, alors la formule de définition de `NB_CLT_CAT12` s'écrit :

```
DEF= Nombre(CLIENT(:CATEGORIE in {1,2}
                and passe:COMMANDE(concerne:LIGNECOM(reference:$))))
```

4.3 Définition syntaxique et sémantique

Dans la suite de ce mémoire, nous serons sans cesse amenés à manipuler les formules de définition. Nous devrons, notamment, réaliser une étude de cohérence des formules. L'introduction intuitive au langage ne suffit alors plus : il faut disposer d'une définition formelle de sa syntaxe et de sa sémantique afin de savoir si une formule est correcte et afin de comprendre sa signification.

La définition formelle du langage fait l'objet de cette section. Pour chacun de ses concepts, nous préciserons la sémantique et la syntaxe. Un résumé complet de la syntaxe du langage en formalisme BNF se trouve en annexe III.

4.3.1 Conventions

- Pour exprimer le fait que le résultat de l'évaluation d'une expression quelconque `expr` fournit un résultat `r`, on dira que `expr` représente (ou désigne) `r`.
- Dans la description de la sémantique, nous utiliserons la convention suivante :

  Si
  - `TE` est un type d'entités quelconque d'un schéma EA déductif,
  - `e` est une entité quelconque de `TE`,
  - `a` est un attribut quelconque de `TE`,

  alors `e.a` représente la valeur de l'attribut `a` pour l'entité `e`.

- On appelle "type d'une expression `expr`", le type de la valeur qui résulte de l'évaluation de `expr`.
- Dans de nombreux cas, nous ne ferons pas de différence, dans nos définitions, entre une expression et le résultat de son évaluation dans un certain contexte.

Exemple : Si `expr` est une expression et `r` le résultat de son évaluation dans un certain contexte et si on veut tester l'appartenance d'une valeur quelconque `x` au résultat `r` de l'évaluation, on écrira souvent `x ∈ expr` alors qu'il serait plus juste d'écrire `x∈r`.

- Voici les conventions BNF utilisées pour la définition de la sémantique et de la syntaxe de chaque construction du langage :

Symbole BNF	Signification
e	Symbole terminal du langage
e	Symbole non terminal du langage
::=	Définit un symbole non terminal du langage
[...]	Elément facultatif
{...}	Elément qui peut être répété de 0 à N fois
...|...|...	Choix d'un élément parmi N

4.3.2 Types et opérations sur les types

Le langage supporte trois types de valeurs : numérique, booléen et chaîne de caractères. Pour chacun de ces trois types, nous précisons quel est l'ensemble des valeurs qui constitue ce type, quelles sont les opérations permises sur ce type et quelle est la règle de construction des constantes appartenant à ce type.

4.3.2.1 Type numérique

L'ensemble des valeurs est l'ensemble des nombres entiers positifs, négatifs ou nuls. Pratiquement, il n'est cependant pas possible de représenter tous les nombres entiers : on se contentera des entiers compris dans un certain intervalle ([-32768,32767] par exemple).

Les opérations possibles sur ce type de valeurs sont l'addition (+), la soustraction (-), la négation (-), la multiplication (*), la division (/) et les différentes opérations de comparaison (=,<=,>=,<,>,<>).

Syntaxe BNF pour la construction des constantes numériques :

```
ConstNum::= [Signe]Chiffre{Chiffre}
Chiffre ::= 1|2|3|4|5|6|7|8|9|0
Signe   ::= -
```

4.3.2.2 Type alphanumérique

L'ensemble des valeurs est l'ensemble des chaînes de caractères (suites de caractères) qu'il est possible de former avec les caractères du code ASCII (codes allant de 32 à 127).

Les seules opérations possibles sur ce type de valeurs sont les différentes opérations de comparaison (=,<=,>=,<,>,<>).

Syntaxe BNF pour la construction des constantes alphanumériques :

```
ConstAlphanum ::= "ChaîneCar"
ChaîneCar ::=     {Caractère}
Caractère ::=     Lettre | Chiffre | CarSpécial
Lettre ::=        LettreMaj | LettreMin
```

```
LettreMaj ::=     A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|
                  V|W|X|Y|Z
LettreMin ::=     a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|
                  v|w|x|y|z
CarSpécial ::=    !| |#|$|%|&|'|(|)|*|+|,|-|.|/|:|;|<|=|>|?|@|[|
                  \|]|^|_|{|}|`|~
```

4.3.2.3 Type booléen

L'ensemble des valeurs est l'ensemble des valeurs de vérité : vrai et faux. Les opérations possibles sur ce type de valeurs sont la conjonction (`and`), la disjonction (`or`), la négation (`not`) ainsi que les deux opérations de comparaison = et <>.

Syntaxe BNF pour la construction des constantes booléennes :

```
ConstBool ::= TRUE | FALSE
```

4.3.3 Nom des éléments d'un schéma

Les formules de définition font référence aux différents éléments d'un schéma EA (type d'entités, type d'associations ou attribut) en les désignant par leur nom. Le nom de chaque élément du schéma EA peut être constitué de lettres (majuscules ou minuscules), de chiffres et du caractère '_'. Il commence obligatoirement par une lettre.

Syntaxe BNF pour la construction du nom des éléments

```
TEntité ::= NomElement
TAssociation ::= NomElement
Attribut ::= NomElement
NomElement ::= Lettre {Lettre | Chiffre} | _ }
```

Les concepts de `TEntité`, `TAssociation` et `Attribut` correspondent respectivement à un nom de type d'entités, un nom de type d'associations et un nom d'attribut du schéma EA déductif.

4.3.4 Formule de définition

4.3.4.1 Sémantique

L'objectif d'une formule de définition est de préciser comment la valeur d'un attribut dérivable doit être calculée à partir de la valeur d'autres éléments d'un schéma EA déductif. Les formules de définition sont précisées au niveau d'un type d'entités et non, comme ce serait plus logique, au niveau des instances (entités) de ce type[3] : quand on donne la formule de définition d'un attribut dérivable d'un type d'entités, on spécifie la manière dont la valeur de cet attribut doit être calculée pour toutes les entités de ce type. Pour personnaliser une formule de définition à chaque entité particulière d'un type, nous utiliserons le concept d'entité courante (cfr. *4.3.9.1*, point 2).

---

[3] Logiquement, il faudrait spécifier la formule de définition d'un attribut dérivable `a` d'un certain type d'entités `TE` pour chaque entité de `TE`. En effet, la manière de calculer la valeur de `a` est spécifique à chaque entité particulière de `TE`. La formule de définition devrait donc être spécifiée au niveau de chaque entité (au niveau des instances) et non au niveau du type d'entités.

Une formule de définition (Fd) se présente ainsi :

```
Fd ::= DEF= Expr
```

où Expr est une expression. Une expression est soit une expression simple, soit une expression arithmétique, soit une expression logique.

Si expr est l'expression constituant la formule de définition d'un attribut dérivable a, la valeur de a pour une entité e est obtenue en évaluant l'expression expr pour l'entité e. Il faut que le type de l'expression expr soit le même que le type de a. De plus, il faut que l'évaluation de expr rende une valeur unique.

Si
- TE est un type d'entités quelconque du schéma EA déductif,
- $\{e_1, \ldots, e_n\}$ est l'ensemble des entités de TE,
- a est un attribut dérivable de TE dont la formule de définition est donnée par expr,
- f(e) est le résultat de l'évaluation de l'expression expr pour l'entité e de TE,

alors $\forall\ i \in [1..n]: e_i.a = f(e_i)$.

4.3.4.2 Syntaxe

```
Fd ::=      DEF=Expr
Expr ::=    ExprSimple | ExprArithm | ExprLog
```

4.3.5 Expression simple

4.3.5.1 Sémantique

Une expression simple est une constante (Constante), une expression de désignation d'un attribut du schéma EA déductif (Attr) ou un appel de fonction (AppelFct). Une expression simple peut être de type numérique, alphanumérique ou booléen.

Une constante est une constante booléenne, numérique ou alphanumérique. Le résultat de l'évaluation d'une constante est la valeur représentée par celle-ci. Les autres types d'expressions simples (Attr, AppelFct) sont décrits dans les paragraphes suivants.

4.3.5.2 Syntaxe

```
ExprSimple ::=    Constante | Attr | AppelFct
Constante ::=     ConstBool | ConstNum | ConstAlphanum
```

4.3.6 Expression arithmétique

4.3.6.1 Sémantique

Une expression arithmétique est constituée de plusieurs expressions simples reliées entre elles par des opérateurs (*,/,+,-). Il faut que toutes les expressions simples figurant dans une expression arithmétique soient de type numérique et qu'elles représentent toutes une valeur unique.

Nous n'entrons pas dans le détail des règles d'évaluation des expressions arithmétiques. Disons simplement qu'elles reposent sur les règles classiques de priorité entre opérateurs et

que l'ordre d'évaluation dicté par ces règles de priorité peut être modifié par l'utilisation de parenthèses.

Une expression arithmétique rend une valeur unique de type numérique. Il faut donc que tout attribut dérivable dont la formule de définition est une expression arithmétique soit de type numérique.

4.3.6.2 Syntaxe

```
ExprArithm ::=    Terme | Terme OpAdditif ExprArithm
Terme ::=         Facteur | Facteur OpMult Terme
Facteur ::=       ExprSimple | - Facteur |(ExprArithm)
OpAdditif ::=     - | +
OpMult ::=        * | /
```

4.3.7 Expression booléenne (ou logique)

4.3.7.1 Sémantique

Une expression booléenne est constituée de plusieurs **propositions atomiques** reliées entre elles par des opérateurs logiques (`and`, `or`, `not`). Une expression atomique est

- soit constituée de deux expressions arithmétiques reliées par un opérateur de comparaison (<,>,<=,>=,=,<>);
- soit constituée de deux expressions simples de même type reliées par un opérateur de comparaison (<,>,<=,>=,=,<>);
- soit constituée d'une seule expression simple de type booléen.

Ici non plus, nous n'entrons pas dans le détail des règles d'évaluation des expressions booléennes. Disons simplement qu'elles reposent sur les règles classiques de priorité entre opérateurs et que l'ordre d'évaluation dicté par ces règles de priorité peut être modifié par l'utilisation de parenthèses.

Chaque proposition atomique et l'expression booléenne complète représentent une valeur unique de type booléen. Il faut donc que tout attribut dérivable dont la formule de définition est une expression booléenne soit de type booléen.

4.3.7.2 Syntaxe

```
ExprLog ::=       Conjonction or ExprLog | Conjonction
Conjonction::=    Négation and Conjonction | Négation
Négation ::=      PropAtom | not Negation | (ExprLog)
PropAtom ::=      ExprArithm | ExprArithm OpComp ExprArithm
OpComp ::=        <|<=|>|>=|=|<>
```

4.3.8 Expression de désignation d'un attribut

4.3.8.1 Sémantique

Une expression de désignation d'un attribut d'un schéma EA déductif (`Attr`) se présente ainsi :

```
Attr ::= EnsEnt.Attribut
```

où

- `EnsEnt` est une expression qui désigne un ensemble d'entités du même type (c'est-à-dire appartenant au même type d'entités);
- `Attribut` est le nom d'un attribut des entités de l'ensemble `EnsEnt`.

`EnsEnt.Attribut` représente l'ensemble des valeurs prises par les entités de l'ensemble `EnsEnt` pour l'attribut `Attribut`. Souvent, l'ensemble `EnsEnt` contient une seule entité : `EnsEnt.Attribut` désigne alors une valeur unique[4] (la valeur d'un seul attribut).

En d'autres mots :

Si
- `E` est un ensemble d'entités appartenant au même type,
- les entités de `E` sont $\{e_1, \ldots, e_n\}$,
- `a` est un attribut quelconque des entités de `E`,

alors $E.a = \{e_1.a, \ldots, e_n.a\}$.

`Attr` est le seul type d'expressions qui peut rendre un ensemble de valeurs. En effet, les autres types d'expressions (expression arithmétique, expression booléenne, constante ou appel de fonction) rendent tous une valeur unique. Suivant le contexte dans lequel `Attr` est utilisé, son évaluation doit rendre une valeur unique (ex : `Attr` utilisé dans une expression arithmétique) ou peut rendre plusieurs valeurs (ex : `Attr` utilisé dans une condition d'appartenance avec l'opérateur `in`). Nous étudions ce problème en détail dans l'étude de cohérence d'un schéma EA déductif (cfr. *7.2.1.3*).

En supposant que les entités de l'ensemble `E` appartiennent au type d'entités `TE`, le type de l'expression `Attr E.a` est le type de l'attribut `a` de `TE` (numérique, booléen ou alphanumérique) : le type d'une expression `Attr` est le type des valeurs désignées par `Attr`.

4.3.8.2 Syntaxe

```
Attr ::= EnsEnt.Attribut
```

[4] Dans la suite de ce travail, on dira que *`Attr` représente une valeur unique* si l'ensemble désigné par `Attr` contient un seul élément et que *`Attr` représente plusieurs valeurs (ou un ensemble de valeurs)* s'il contient un élément ou plus. Afin de simplifier l'exposé, nous écartons pour l'instant la possibilité pour une expression de type `Attr` de désigner 0 valeur : nous analyserons cette éventualité dans la suite (cfr. *4.4*)

4.3.9 Expression de désignation d'un ensemble d'entités

4.3.9.1 Sémantique

Un ensemble d'entités `EnsEnt` peut prendre différentes formes :

1. Désignation de toutes les entités d'un type
Dans ce cas, l'expression `EnsEnt` se présente ainsi :

```
EnsEnt ::= TEntité
```

où `TEntité` est le nom d'un type d'entités du schéma EA déductif.

`TEntité` représente l'ensemble de toutes les entités du type `TEntité` : le nom d'un type d'entités représente l'ensemble de toutes les entités de ce type.

2. Désignation de l'entité courante
Dans ce cas, l'expression `EnsEnt` se présente ainsi :

```
EnsEnt ::= $
```

Quand on donne la formule de définition d'un attribut d'un type d'entités, on spécifie la manière dont la valeur de cet attribut doit être calculée pour toutes les entités de ce type. La formule doit donc être personnalisable à chaque entité particulière. Pour ce faire, on utilise le concept d'entité courante.

Dans la formule de définition d'un attribut dérivable `a` d'un type d'entités `TE`, l'entité courante est l'entité de `TE` pour laquelle on est en train de calculer la valeur de `a`. On y fait référence en utilisant le symbole '`$`'. Le symbole `$` permet une généralisation des formules de définition : on a une seule formule de définition pour un attribut dérivable d'un type `TE` et non une formule spécifique pour chaque entité de `TE`. La formule de définition générale peut ensuite être personnalisée à chaque entité particulière de `TE`.

En d'autres mots :
Si - on donne la formule de définition d'un attribut `a` d'un type d'entités `TE`,
- les entités de `TE` sont $\{e_1, \ldots, e_n\}$,

alors le symbole `$` qui peut apparaître dans la formule de définition de `a` représente
l'entité e_1 quand on calcule la valeur de e_1.a
l'entité e_2 quand on calcule la valeur de e_2.a
...
l'entité e_n quand on calcule la valeur de e_n.a

3. Désignation d'un ensemble d'entités respectant une certaine condition
Dans ce cas, l'expression `EnsEnt` se présente ainsi :

```
EnsEnt ::= TEntité(Csel)
```

où
- `TEntité` est le nom d'un type d'entités du schéma EA déductif (`TEntité` désigne donc l'ensemble de toutes les entités de ce type);
- `Csel` est une condition de sélection portant sur l'ensemble des entités de `TEntité`.

`TEntité(Csel)` représente l'ensemble des entités de `TEntité` qui respectent la condition `Csel`.

En d'autres mots :
Si
- `TE` est un type d'entités quelconque du schéma EA déductif,
- $\{e_1, \ldots, e_n\}$ est l'ensemble des entités de `TE`,
- `cd` est une condition de sélection portant sur $\{e_1, \ldots, e_n\}$,

alors $\forall\ i \in [1..n] : e_i \in TE(cd)$ ssi[5] e_i respecte la condition `cd`.

4.3.9.2 Syntaxe

```
EnsEnt ::= TEntité | $  | TEntité(Csel)
```

4.3.10 Condition de sélection

4.3.10.1 Sémantique

Une condition de sélection (`Csel`) est
- soit constituée de plusieurs conditions d'association reliées par des opérateurs logiques (`and`/`or`/`not`);
- soit une condition d'association unique.

4.3.10.2 Syntaxe

```
Csel ::=     CselConj | CselConj or Csel
CselConj ::= CselNeg  | CselNeg  and CselConj
CselNeg ::=  Cass     | not CselNeg | (Csel)
```

4.3.11 Condition d'association

4.3.11.1 Sémantique

Une condition d'association portant sur un type d'entités `TE` concerne, comme son nom l'indique, l'association qui existe entre les entités de `TE` et d'autres éléments du schéma EA déductif. Analysons les éléments qui peuvent être associés à une entité particulière `e` de `TE` :

- Une première association possible est celle qui associe `e` à d'autres entités $e_1, \ldots, e_n$ étant respectivement de type $TE_1, \ldots, TE_n$. Une telle association est possible à condition qu'il existe dans le schéma EA déductif un type d'associations ta_i entre `TE` et chacun des types $TE_1, \ldots, TE_n$. Il existe donc une association explicite[6] entre `e` et chacune des entités $e_1, \ldots, e_n$.
- Une seconde association possible est celle qui relie `e` aux valeurs de ses attributs. `TE` est, en effet, caractérisé par des attributs $a_1, \ldots, a_p$ et chaque entité de `TE` (et donc, en particulier, `e`) est caractérisée par des valeurs de ces attributs. Il existe donc une association implicite[7] entre `e` et la valeur que prend chacun de ses attributs.

---

[5] ssi est l'abréviation de "si et seulement si".

[6] L'association est explicite parce que le type d'associations ta_i reliant `TE` à TE_i est explicitement indiqué sur le schéma EA déductif.

[7] L'association est implicite parce que, sur le schéma EA déductif, rien n'indique explicitement cette association.

Comme il existe deux espèces d'associations possibles, nous distinguons deux types de conditions d'association portant sur l'ensemble des entités d'un type `TE` :

- les conditions portant sur l'association entre les entités de `TE` et les entités d'un autre ensemble d'entités via des associations du schéma EA déductif (associations explicites) : condition d'association avec des entités;
- les conditions portant sur l'association entre les entités de `TE` et la valeur que prend chacun des attributs de ces entités (associations implicites) : condition d'association avec des valeurs d'attribut.

Décrivons à présent les deux types de conditions d'association.

1. Condition d'association avec des entités

Dans ce cas, la condition d'association (`Cass`) se présente ainsi :

```
Cass ::= TAssociation:EnsEnt
```

Pour donner une signification à `TAssociation` et à `EnsEnt`, il faut replacer la condition d'association `Cass` dans son contexte d'utilisation c'est-à-dire en tant que condition de sélection `Csel` (cfr. *4.3.9.1*, point *3*). La définition de `EnsEnt` s'écrivait alors ainsi :

```
EnsEnt ::= TEntité(Csel)
```

En considérant à présent que la condition de sélection `Csel` est constituée d'une seule condition d'association `Cass`, la forme de `EnsEnt` peut se réécrire ainsi[8]:

$$\texttt{EnsEnt ::= TEntité(TAssociation:EnsEnt}_1\texttt{)}$$

Donnons tout d'abord la signification des différents intervenants :

- `TAssociation` est le nom d'un type d'associations du schéma EA déductif reliant l'ensemble des entités de `TEntité` à l'ensemble des entités $\texttt{EnsEnt}_1$;
- $\texttt{EnsEnt}_1$ désigne un ensemble d'entités du même type (cfr. *4.3.9.1*).

$\texttt{TEntité(TAssociation:EnsEnt}_1\texttt{)}$ représente l'ensemble des entités de `TEntité` reliées via le type d'associations `TAssociation` à au moins une entité de $\texttt{EnsEnt}_1$. L'expression $\texttt{EnsEnt}_1$ peut être, elle-même, d'une des trois formes spécifiées en *4.3.9.1*.

En d'autres mots :

Si
- `TE1` et `TE2` sont deux types d'entités du schéma EA déductif reliés par un type d'associations `ta`,
- $\{\texttt{e1}_1, \ldots, \texttt{e1}_n\}$ est l'ensemble des entités de `TE1`,
- `EnsEnt` est une expression qui désigne un sous-ensemble des entités de `TE2` : $\alpha = \{\texttt{e2}_1, \ldots, \texttt{e2}_p\}$ est l'ensemble des entités désigné par `EnsEnt`,
- β est l'ensemble des associations de type `ta` reliant une entité de `TE1` à une entité de α. β se présente sous la forme d'un ensemble de couples : si $\texttt{e1}_i \in \texttt{TE1}$ et $\texttt{e2}_j \in \alpha$, alors $(\texttt{e1}_i, \texttt{e2}_j) \in \beta$ ssi il existe une association de type `ta` reliant $\texttt{e1}_i$ et $\texttt{e2}_j$,

[8] L'utilisation d'indices permet d'éviter toute confusion entre les différents intervenants.

alors $\forall$ i$\in$[1..n]: $e1_i \in$ TE1(ta:EnsEnt)
ssi il existe au moins une entité $e2_j \in \alpha$ telle que $(e1_i, e2_j) \in \beta$.

Remarque : Dans un type d'associations récursif ta portant sur le type d'entités TE, la condition d'association se présente ainsi : TE(ta:TE(...)). Il est, par conséquent, impossible de savoir dans quel sens parcourir ta. Pour remédier à ce problème, le langage permet de remplacer le nom d'un type d'entités dans une condition de sélection par le nom du rôle joué par ce type d'entités dans le type d'associations.

2. Condition d'association avec des valeurs d'attribut
Dans ce cas, la condition d'association (Cass) se présente ainsi :

```
Cass ::= TAssociation:Attribut Capp
```

Pour donner une signification à TAssociation, à Attribut et à Capp, il faut de nouveau replacer la condition d'association Cass dans son contexte d'utilisation (cfr. *4.3.9.1*, point *3*). En considérant que la condition de sélection Csel est constituée d'une seule condition d'association Cass, la forme de EnsEnt peut se réécrire ainsi [9]:

```
EnsEnt ::= TEntité(:Attribut Capp)
```

Donnons tout d'abord la signification des différents intervenants :

- Attribut est le nom d'un attribut de TEntité;
- Capp est une condition d'appartenance portant sur l'attribut Attribut de TEntité.

TEntité(:Attribut Capp) représente l'ensemble des entités de TEntité pour lesquelles la valeur de l'attribut Attribut respecte la condition d'appartenance Capp.

En d'autres mots :
Si
- TE est un type d'entités du schéma EA déductif,
- $\{e_1, \ldots, e_n\}$ est l'ensemble des entités de TE,
- a est un attribut de TE,
- cd est une condition d'appartenance portant sur a,

alors $\forall$ i$\in$[1..n]: $e_i \in$ TE(:a cd) ssi e_i.a respecte la condition cd.

4.3.11.2 Syntaxe

```
Cass ::=          CassEntité | CassAttribut
CassEntité ::=    (TAssociation:EnsEnt)
CassAttribut ::=  (:Attribut Capp)
```

[9] Remarquons que le nom de l'association entre TEntité et Attribut est absent puisqu'il n'existe pas d'association explicite entre une entité et la valeur de ses attributs.

4.3.12 Condition d'appartenance

4.3.12.1 Sémantique

La forme générale d'une condition d'appartenance est la suivante :

```
Capp ::= Rel EnsVal
```

où

- `Rel` est une relation du type <,>,=,<=,>=,<>, `in`, `not in`;
- `EnsVal` est soit un ensemble d'expressions simples (`{ExprSimple}`)
  soit une expression simple unique (`ExprSimple`).

Rappelons que la seule forme d'expression simple susceptible de rendre, lors de son évaluation, plusieurs valeurs est l'expression de désignation d'un attribut `Attr`.

Nous avons vu, en considérant que la condition de sélection `Csel` est constituée d'une seule condition d'association `Cass`, que `EnsEnt` peut s'écrire (cfr. *4.3.11.1*, point *2*) :

```
EnsEnt ::= TEntité(:Attribut Capp)
```

En remplaçant à présent `Capp` par sa définition, on obtient :

```
EnsEnt ::= TEntité(:Attribut Rel EnsVal)
```

Envisageons les différents cas possibles pour `EnsVal` :

1. `EnsVal` représente une valeur unique

Dans ce cas, `Rel` est un des opérateurs de comparaison suivants : =,<,>,<=,>=,<>

Si
- on considère que `Rel` vaut '=' [10],
- `TE` est un type d'entités du schéma EA déductif,
- $\{e_1, \ldots, e_n\}$ est l'ensemble des entités de `TE`,
- `a` est un attribut de `TE`,
- `expr` est une expression simple rendant une valeur unique,
- α est la valeur unique rendue par l'évaluation de l'expression `expr`,

alors

$\forall\ i \in [1..n]:\ e_i \in$ `TE(:a=expr)`
ssi $e_i.a = \alpha$.

2. `EnsVal` représente un ensemble de valeurs (plusieurs valeurs)

Dans ce cas, `Rel` est un des opérateurs ensemblistes suivants : `in`, `not in`

Un ensemble de valeurs peut se présenter sous deux formes :

- soit sous la forme $\{expr_1, \ldots, expr_n\}$ où chaque $expr_i$ est une expression simple rendant une valeur unique ou un ensemble de valeurs (1^er^ cas);

[10] On peut tenir le même raisonnement pour les autres relations <,>,<=,>= ou <>.

- soit sous la forme `expr` où `expr` est une expression simple rendant un ensemble de valeurs (2ème cas). La seule forme d'expression qui peut rendre un ensemble de valeurs est l'expression de désignation d'un attribut `Attr`.

1er cas

Si
- on considère que `Rel` vaut `'in'` [11],
- `TE` est un type d'entités du schéma EA déductif,
- $\{e_1, \ldots, e_n\}$ est l'ensemble des entités de `TE`,
- `a` est un attribut de `TE`,
- $\{expr_1, \ldots, expr_k\}$ est un ensemble d'expressions simples,
- $\alpha_1, \ldots, \alpha_p$ sont les valeurs rendues par l'évaluation des expressions $expr_1, \ldots, expr_k$,

alors

$\forall\ i \in [1..n]$: $e_i \in$ TE(:a in $\{expr_1, \ldots, expr_k\}$)
ssi e_i.a $\in \{\alpha_1, \ldots, \alpha_p\}$.

2ème cas

Si
- on considère que `Rel` vaut `'in'` [11],
- `TE` est un type d'entités du schéma EA déductif,
- $\{e_1, \ldots, e_n\}$ est l'ensemble des entités de `TE`,
- `a` est un attribut de `TE`,
- `attr` est une expression de désignation d'un attribut rendant plusieurs valeurs,
- $\alpha_1, \ldots, \alpha_p$ sont les valeurs rendues par l'évaluation de l'expression `attr`,

alors

$\forall\ i \in [1..n]$: $e_i \in$ TE(:a in attr)
ssi e_i.a $\in \{\alpha_1, \ldots, \alpha_p\}$.

4.3.12.2 Syntaxe

```
Capp ::= Rel EnsVal
Rel ::= OpComp | OpEnsembliste
OpEnsembliste ::= in | not in
EnsVal ::= ExprSimple | {ExprSimple{,ExprSimple}}
```

4.3.13 Les fonctions

4.3.13.1 Sémantique

Le langage d'expression des formules de définition possède des fonctions prédéfinies : fonctions `Somme`, `Min`, `Max`, `NombreVal`, `Moyenne` et `Nombre`.

La forme générale des appels de fonction est la suivante :

```
Fct(Arguments)
```

où
- `Fct` est le nom de la fonction appelée;
- `Arguments` est la suite des arguments de la fonction.

[11] On peut tenir le même raisonnement pour la relation `not in`.

Dans la formule de définition d'un attribut dérivable, l'expression `Fct(Arguments)` représente la valeur retournée par la fonction `Fct` exécutée avec les arguments `Arguments`. Chaque appel de fonction rend au maximum une valeur [12]. Les fonctions `Somme`, `Min`, `Max`, `Moyenne` et `NombreVal` possèdent une syntaxe similaire : nous les spécifions ensemble.

1. Fonctions `Somme`, `Min`, `Max`, `Moyenne` et `NombreVal`

Ces fonctions se présentent ainsi :

```
FctSomme     ::= Somme(I=EnsEnt;Expr_I)
FctMin       ::= Min(I=EnsEnt;Expr_I)
FctMax       ::= Max(I=EnsEnt;Expr_I)
FctMoyenne   ::= Moyenne(I=EnsEnt;Expr_I)
FctNombreVal ::= NombreVal(I=EnsEnt;Expr_I)
```

où

- `EnsEnt` est une expression représentant un ensemble d'entités du même type sur lequel la fonction porte (cfr. *4.3.9.1*);
- `I` est un artifice utilisé pour nommer individuellement chaque entité de l'ensemble `EnsEnt` : `I` est comparable à un indice. Il s'agit d'une entité fictive qui, dans la fonction, s'assimile successivement à chaque entité de `EnsEnt`;
- $Expr_I$ est une expression rendant une valeur unique pour chaque entité de `EnsEnt` : $Expr_I$ dépend de l'indice `I`. Dans $Expr_I$ on utilise l'entité fictive `I` pour désigner individuellement les entités de l'ensemble `EnsEnt` chacune à son tour.

Les fonctions `Somme`, `Min`, `Max`, `Moyenne` et `NombreVal` calculent la valeur de l'expression $Expr_I$ pour chaque entité `I` de l'ensemble `EnsEnt` et renvoient respectivement la somme, le minimum, le maximum, la moyenne de ces valeurs ou le nombre de valeurs distinctes.

Pour les fonctions `Somme` et `Moyenne`, l'expression $Expr_I$ doit être de type numérique : $Expr_I$ doit être une expression arithmétique ou une expression simple de type numérique. Une expression `Somme(I=EnsEnt;`$Expr_I$`)` ou `Moyenne(I=EnsEnt;`$Expr_I$`)` est toujours de type numérique.

Pour les fonctions `Min` et `Max`, l'expression $Expr_I$ peut être de type numérique ou alphanumérique : $Expr_I$ doit être une expression arithmétique, une expression simple de type numérique ou une expression simple de type alphanumérique. Une expression `Min(I=EnsEnt;`$Expr_I$`)` ou `Max(I=EnsEnt;`$Expr_I$`)` est de type numérique si $Expr_I$ est de type numérique et de type alphanumérique si $Expr_I$ est de type alphanumérique.

Pour la fonction `NombreVal`, l'expression $Expr_I$ peut être de n'importe quel type : $Expr_I$ peut être une expression arithmétique, booléenne ou simple. Une expression `NombreVal(I=EnsEnt;`$Expr_I$`)` est toujours de type numérique.

---

[12] Il est possible qu'une fonction renvoie 0 valeur. Nous analyserons cette éventualité en détail au point *4.4*.

Attention : Les différentes expressions simples qui figurent dans l'expression $Expr_I$ peuvent être des expressions de désignation d'un attribut (`Attr`) ou des constantes (`Constante`) mais pas des appels de fonction.

Si
- $E=\{e_1, \ldots, e_n\}$ est un ensemble d'entités,
- ex_I désigne une expression dépendant de l'entité fictive `I` (`I` joue le rôle d'un indice qui, dans la fonction, s'assimile successivement aux entités $e_1, \ldots, e_n$),
- α_{ei} est le résultat de l'évaluation de ex_{ei} où ex_{ei} désigne l'expression ex_I dans laquelle on a remplacé l'entité fictive `I` par l'entité réelle e_i,

alors

`Somme(I=E;`ex_I`)`	= $\alpha_{e1} + \ldots + \alpha_{en}$
`Min(I=E;`ex_I`)`	= minimum des valeurs $\alpha_{e1}, \ldots, \alpha_{en}$
`Max(I=E;`ex_I`)`	= maximum des valeurs $\alpha_{e1}, \ldots, \alpha_{en}$
`Moyenne(I=E;`ex_I`)`	= moyenne des valeurs $\alpha_{e1}, \ldots, \alpha_{en}$
`NombreVal(I=E;`ex_I`)`	= nombre de valeurs distinctes parmi $\alpha_{e1}, \ldots, \alpha_{en}$

Dans le point *4.3.9.1*, nous avons vu que l'expression de désignation d'un ensemble d'entités (`EnsEnt`) pouvait être constituée soit d'un type d'entités du schéma EA (sur lequel on applique ou non une condition de sélection), soit de l'entité courante (`$`).

Dans les expressions $Expr_I$ figurant dans une fonction `Somme`, `Min`, `Max`, `Moyenne` ou `NombreVal`, il faut ajouter une possibilité supplémentaire : il faut pouvoir désigner l'entité fictive `I` en tant que `EnsEnt`. `I` est aussi une expression de désignation d'un ensemble d'entités (ensemble constitué d'un seul élément). On doit, dès lors, réécrire la définition de `EnsEnt` comme suit :

```
EnsEnt ::= TEntité | $ | TEntité(Csel) | I
```

2. Fonction `Nombre`

La fonction `Nombre` se présente ainsi :

```
FctNombre ::= Nombre(EnsEnt)
```

où `EnsEnt` est une expression représentant un ensemble d'entités (cfr. *4.3.9.1*).

La fonction `Nombre` renvoie le nombre d'entités distinctes comprises dans `EnsEnt`. Une expression `Nombre(EnsEnt)` est toujours de type numérique.

4.3.13.2 Syntaxe

```
AppelFct      ::= FctSomme     | FctMin      | FctMax |
                  FctNombre    | FctMoyenne  | FctNombreVal
FctSomme      ::= Somme(I=EnsEnt;ExprI)
FctMin        ::= Min(I=EnsEnt;ExprI)
FctMax        ::= Max(I=EnsEnt;ExprI)
FctMoyenne    ::= Moyenne(I=EnsEnt;ExprI)
FctNombreVal  ::= NombreVal(I=EnsEnt;ExprI)
FctNombre     ::= Nombre(EnsEnt)
EnsEnt ::= TEntité | $ | TEntité(Csel) | I
```

4.4 Valeur NULL

Nous avons déjà évoqué la possibilité pour certaines expressions simples de représenter 0 valeur. Ceci peut être le cas pour les expressions de désignation d'un attribut (`Attr`) et pour les fonctions agrégatives (`AppelFct`).

Par exemple, en se basant sur le schéma de la figure 4-1, l'expression `Attr`

```
COMMANDE(:MONTANT<100 and :MONTANT>200).DATE
```

ne représente aucune valeur (l'ensemble des valeurs d'attribut désigné par l'expression est vide). De même, en supposant qu'aucun client n'habite Namur, l'expression

```
Moyenne(I=CLIENT(:LOCALITE="Namur");I.COMPTE)
```

ne représente aucune valeur.

Par conséquent :

- une expression simple de type `Attr` peut représenter 0, 1 ou plusieurs valeurs;
- une expression simple de type `AppelFct` peut représenter 0 ou 1 valeur.

Quand une expression simple représente 0 valeur, on dit que cette expression prend la valeur `NULL` : `NULL` représente l'absence de valeur. Dans le même ordre d'idée, si pour une entité `e` d'un type `TE`, un attribut facultatif `a` n'a pas de valeur, on dira que `a` prend la valeur `NULL` pour `e`.

Analysons l'effet d'une valeur `NULL` dans le calcul de la valeur d'un attribut dérivable : la formule de définition d'un attribut dérivable est constituée d'une expression `expr` constituée, à son tour, de plusieurs expressions simples $expr_1, \ldots, expr_n$ reliées entre elles par des opérateurs. Si, lors de l'évaluation de `expr`, une des expressions simples $expr_1, \ldots, expr_n$ prend la valeur `NULL`, `expr` prend aussi la valeur `NULL` : il est, en effet, impossible de déterminer la valeur de `expr` puisqu'un de ses opérandes n'a pas de valeur.

Précisons le comportement des fonctions agrégatives `Somme`, `Min`, `Max`, `Moyenne` et `NombreVal` vis-à-vis de la valeur `NULL`[13]. Dans les expressions

```
Somme(I=E;expr_I)
Min(I=E;expr_I)
Max(I=E;expr_I)
Moyenne(I=E;expr_I)
NombreVal(I=E;expr_I)
```

- si `E` représente un ensemble de 0 entité, alors `Somme(I=E;expr_I)` `Min(I=E;expr_I)`, `Max(I=E;expr_I)` et `Moyenne(I=E;expr_I)` prennent la valeur `NULL`. `NombreVal(I=E;expr_I)` prend la valeur 0;
- si l'expression $expr_I$ prend la valeur `NULL` pour une entité $e \in E$, alors les fonctions ignorent la valeur de $expr_I$ associée à `e` dans leur calcul[14].

[13] La fonction `Nombre` n'est pas concernée par la valeur `NULL` : si `E` est un ensemble de 0 entité, `Nombre(E)` vaut 0.

4.5 Limitation du langage

Quoique relativement puissant, le langage d'expression des formules de définition, présente certaines lacunes. En effet, il ne permet pas d'exprimer directement (en une seule formule de définition) toutes les requêtes qui pourraient surgir dans l'esprit du concepteur du schéma EA déductif. Dans cette section, nous présentons la limitation principale du langage et nous expliquons comment contourner cette limitation. Nos exemples se basent sur le schéma EA déductif de la figure 4-1.

Considérons la requête suivante : on aimerait définir un attribut dérivable NB_PROD_SUP de FOURNISSEUR qui représente le nombre de produits d'un fournisseur dont le prix de vente est supérieur à 200 et dont la quantité en stock est supérieure à la quantité totale commandée.

Il s'agit donc, pour un fournisseur donné, de compter un certain nombre de produits. Un squelette de formule de définition ressemblerait à ceci :

```
DEF= Nombre(PRODUIT(livre:$ and ...)
```

Ces produits sont soumis à une condition de sélection : il faut, d'une part, que les produits retenus aient un prix supérieur à 200 et, d'autre part, que la quantité présente en stock de ces produits soit supérieure à la quantité totale commandée de ceux-ci. Si la première partie de la condition ne pose aucun problème, on ne peut pas en dire autant de la seconde. Il faudrait, en effet, pour un produit donné, compter la quantité totale commandée pour ce produit. Essayons de compléter la requête :

```
DEF= Nombre(PRODUIT(livre:$
                and :PRIX_VENTE>=200
                and :QTE_STOCK > Somme(I=LIGNECOM(reference:?);I.QCOM)
```

Une ligne de commande doit être sélectionnée dans la somme si elle fait référence au produit en question. Il faudrait donc, à la place du '?', pouvoir désigner l'entité courante de PRODUIT. Or, ceci est impossible : la seule entité courante qu'il est possible de désigner est celle de FOURNISSEUR c'est-à-dire celle qui correspond au type d'entités contenant l'attribut dérivable qu'on est en train de définir.

La principale lacune du langage réside donc dans le fait que le concept d'entité courante n'est pas suffisamment puissant. Il faudrait disposer d'un mécanisme permettant de désigner l'entité courante de chaque type d'entités "intermédiaire" figurant dans une condition de sélection. Par exemple, dans une expression telle que

```
CLIENT(passe:COMMANDE(concerne:LIGNECOM(reference:PRODUIT(...))))
```

il serait intéressant de pouvoir désigner, dans les conditions de sélection, l'entité courante de CLIENT, de COMMANDE, de LIGNECOM et de PRODUIT.

Pour contourner cette limitation, il faut créer un attribut dérivable "intermédiaire" assimilable à une variable intermédiaire dans les langages de programmation classiques. Dans notre exemple, on définit l'attribut dérivable DIFF_STOCK de PRODUIT qui

---

[14] Si l'expression $expr_I$ prend la valeur NULL pour toutes les entités de E, alors la fonction toute entière prend la valeur NULL.

représente la différence entre la quantité en stock de ce produit et la quantité totale commandée de ce produit :

```
DEF= $.QTE_STOCK - Somme(I=LIGNECOM(reference:$);I.QCOM)
```

On peut ensuite donner la formule de définition de NB_PROD_SUP de FOURNISSEUR :

```
DEF= Nombre (PRODUIT(   FOURNIT:$
                        and :PRIX_VENTE>=200
                        and :DIFF_STOCK > 0))
```

En toute généralité, on peut énoncer la règle suivante : à chaque fois qu'il s'agit de désigner l'entité courante d'un type d'entités intermédiaire TE dans une condition de sélection, il est nécessaire de créer un attribut dérivable intermédiaire dans le type d'entités TE.

Considérons un autre exemple : on aimerait définir l'attribut dérivable COM_IMPAYEE de CLIENT qui représente le nombre de commandes que le client n'a pas honorées entièrement (MONTANT_PAYE < MONTANT). La formule de définition pourrait s'écrire :

```
DEF= Nombre(COMMANDE(passe:$ and :MONTANT_PAYE < ?.MONTANT ))
```

Il est impossible de désigner l'entité courante du type d'entités intermédiaire COMMANDE. On définit par conséquent un attribut dérivable intermédiaire MONTANT_DU de COMMANDE représentant le montant dû sur une commande :

```
DEF= $.MONTANT - $.MONTANT_PAYE
```

On peut ensuite donner la formule de définition de COM_IMPAYEE de CLIENT :

```
DEF= Nombre(COMMANDE(passe:$ and :MONTANT_DU > 0))
```

5. *Résolution de problèmes*

5.1 Introduction

Une fois le modèle EA déductif défini, on peut envisager maintenant d'exploiter celui-ci. La **résolution de problèmes** a pour objectif de rendre un schéma EA déductif **exécutable** : on donne des valeurs à certains attributs et on essaye de calculer les valeurs des autres attributs du schéma à partir des formules de définition. On "exécute" le schéma sur des données réelles.

On désigne par **attributs données**, les attributs du schéma dont les valeurs sont connues avant l'exécution du schéma [15] (ce sont les données du problème), et par **attributs résultats**, les attributs du schéma dont les valeurs sont inconnues et doivent être calculées (ce sont les résultats du problème). L'**exécution d'un schéma** consiste donc à rechercher les valeurs des attributs résultats à partir des valeurs prises par les attributs données sur base des formules de définition du schéma.

Le modèle EA déductif, tel qu'il vient d'être présenté, est neutre vis-à-vis des différentes exploitations qu'on peut en faire : un schéma EA déductif est indépendant par rapport à la manière de le rendre exécutable. Nous envisageons, dans ce chapitre, trois modes d'exploitation particuliers du modèle (il en existe certainement d'autres) :

- calcul des valeurs des attributs en système définitionnel (cfr. *5.2*);
- calcul des valeurs des attributs en système relationnel (cfr. *5.3*);
- calcul des valeurs des attributs en optimisation (cfr. *5.4*).

Remarque : Dorénavant, pour désigner l'attribut `a` d'un type d'entités `TE` on utilisera la notation[16] `TE.a`.

5.2 Système définitionnel

Ce mode d'exploitation est le plus simple : on considère ici que les formules qui définissent les attributs dérivables sont des **définitions**. Il s'agit d'une approche **directionnelle** : les attributs résultats et les attributs données sont fixés à l'avance (on sait à l'avance quels sont les attributs dont les valeurs vont être données et quels sont les attributs dont il va falloir calculer les valeurs).

Les attributs résultats du schéma sont ceux qui sont définis par une formule de définition : ce sont les attributs dérivables du schéma. Les attributs données sont les attributs de base (leurs valeurs doivent obligatoirement être connues au moment de l'exécution du schéma). L'exécution d'un schéma en système définitionnel consiste donc à calculer les valeurs des attributs dérivables à partir des valeurs des attributs de base grâce aux formules de définition.

---

[15] Dans le cas d'un attribut facultatif, la valeur `NULL` est considérée ici comme une valeur classique, au même titre que les autres : s'il est établi qu'une entité prend la valeur `NULL` pour un attribut facultatif, cette valeur est considérée connue.

[16] Il ne faut pas confondre cette notation avec l'expression de désignation d'un ensemble de valeurs d'attribut `TE.a` qui représente l'ensemble des valeurs prises par toutes les entités de `TE` pour l'attribut `a`.

Considérons, par exemple, le schéma EA déductif suivant (les attributs dérivables sont indiqués en italique):

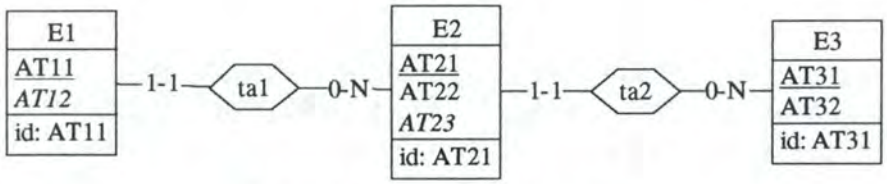


Figure 5-1 : Schéma EA déductif

Les formules de définition sont les suivantes :

- L'attribut `E1.AT12` est défini par `DEF= E2(ta1:$).AT23 * 4`
- L'attribut `E2.AT23` est défini par `DEF= $.AT22 + E3(ta2:$).AT32`

Lors de l'exécution de ce schéma, on calcule les valeurs des attributs `E1.AT12` et `E2.AT23` (attributs dérivables) sur base des valeurs prises par les attributs `E2.AT22` et `E3.AT32` (attributs de base).

Pour exécuter ce schéma, il est nécessaire auparavant de l'instancier :

Soit $\{e1_1, e1_2, e1_3\}$ l'ensemble des entités de `E1`;
$\{e2_1, e2_2, e2_3\}$ l'ensemble des entités de `E2`;
$\{e3_1, e3_2\}$ l'ensemble des entités de `E3`.

Le type d'associations `ta1` est constitué d'un ensemble de couples $(e1_i, e2_j)$ où $e1_i \in$ `E1` et $e2_j \in$ `E2`. Soit $\{(e1_1, e2_1), (e1_2, e2_1), (e1_3, e2_2)\}$ l'ensemble des associations de `ta1`;

De même, le type d'associations `ta2` est constitué d'un ensemble de couples $(e2_i, e3_j)$ où $e2_i \in$ `E2` et $e3_j \in$ `E3`. Soit $\{(e2_1, e3_1), (e2_2, e3_2), (e2_3, e3_2)\}$ l'ensemble des associations de `ta2`.

Graphiquement, on peut représenter ces associations comme suit :

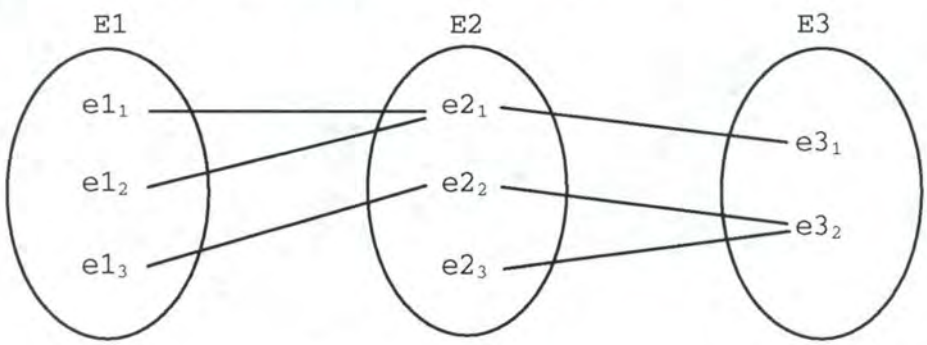


Figure 5-2 : Représentation des associations

Le tableau suivant donne, sur base d'un jeu de valeurs pour les attributs de base, les valeurs correspondantes des attributs dérivables :

Valeurs des attributs de base	**Attributs**	**Valeurs des attributs dérivables**
	$e1_1$.AT12	96
	$e1_2$.AT12	96
	$e1_3$.AT12	60
15	$e2_1$.AT22	
13	$e2_2$.AT22	
19	$e2_3$.AT22	
	$e2_1$.AT23	24
	$e2_2$.AT23	15
	$e2_3$.AT23	21
9	$e3_1$.AT32	
2	$e3_2$.AT32	

Tableau 5-1 : Jeu de valeurs

Dans cet exemple, il est nécessaire de calculer les valeurs prises par `E2.AT23` avant de pouvoir calculer les valeurs de `E1.AT12` puisque `E2.AT23` intervient dans le calcul de `E1.AT12` (on dira, plus tard, que `E1.AT12` dépend de `E2.AT23`).

Pratiquement, l'exploitation d'un schéma EA déductif en système définitionnel peut prendre plusieurs formes :

1. génération d'une feuille de calcul dans un tableur (EXCEL, Lotus, TKSolver, etc.);
2. génération de vues SQL : on transforme le schéma EA déductif en tables SQL. Les attributs de ces tables sont les attributs de base du schéma (les attributs dérivables ne figurent pas dans les tables). Chaque attribut dérivable donne naissance à une vue permettant de calculer ses valeurs.

   Exemple : Le schéma EA déductif de la figure 5-1 est transformé en tables SQL par les définitions SQL suivantes (on reprend, dans les tables, uniquement les attributs de base) :

```
CREATE TABLE E1 (    AT11 integer not null,
                     AT21 integer not null,
                     primary key (AT11),
                     foreign key (AT21) references E2);

CREATE TABLE E2 (    AT21 integer not null,
                     AT22 integer not null,
                     AT31 integer not null,
                     primary key (AT21),
                     foreign key (AT31) references E3);

CREATE TABLE E3 (    AT31 integer not null,
                     AT32 integer not null,
                     primary key (AT31));
```

On définit ensuite la vue E2_AT23 pour calculer les valeurs de E2.AT23 [17]:

```
CREATE VIEW E2_AT23(AT21, AT23)
 AS
SELECT E2.AT21, E2.AT22 + E3.AT32
FROM E2, E3
WHERE E2.AT31 = E3.AT31;
```

On définit finalement la vue E1_AT12 pour calculer les valeurs de E1.AT12 :

```
CREATE VIEW E1_AT12(AT11, AT12)
 AS
SELECT E1.AT12, E2_AT23.AT23 * 4
FROM E1, E2_AT23
WHERE E1.AT21 = E2_AT23.AT21;
```

3. génération de tables SQL : on transforme le schéma EA déductif en tables SQL. Les attributs de ces tables sont les attributs de base du schéma. Pour chaque table T, on crée ensuite une table complémentaire T' dans laquelle on vient ranger toutes les valeurs des attributs dérivables du type d'entités correspondant à T.

   Exemple : Les tables E1, E2 et E3 sont créées comme ci-dessus. Au lieu de définir des vues pour le calcul des attributs dérivables, on crée maintenant des tables complémentaires qui vont être garnies avec les valeurs des attributs dérivables [18]:

```
CREATE TABLE COMP_E1 (  AT11 integer not null,
                        AT12 integer not null,
                        primary key (AT11) );

CREATE TABLE COMP_E2 (  AT21 integer not null,
                        AT23 integer not null,
                        primary key (AT21) );
```

   Il n'est pas nécessaire de créer une table complémentaire pour E3 puisque le type d'entités E3 du schéma possède aucun attribut dérivable. Le calcul proprement dit des valeurs des attributs dérivables ainsi que le stockage de celles-ci dans les tables complémentaires peut se faire à l'aide d'un programme C, PASCAL, COBOL, etc. ou encore à l'aide d'un ensemble de requêtes SQL (script SQL).

4. génération de *Triggers* : un *Trigger* est une **action** que l'on peut associer à une modification d'état dans la base de données (**événement**) lorsque une certaine **condition** est vérifiée. En reprenant le schéma de la figure 5-1, on pourrait associer à la table correspondant à E3, le *Trigger* suivant :

   Evénement : modification de l'attribut AT32 d'une entité (tuple) $e3_i$ de E3.
   Action : modifier la valeur de l'attribut AT23 de chaque entité $e2_j$ de E2 en fonction de la nouvelle valeur de $e3_i$.AT32.
   Condition : $e2_j$ doit être associée à $e3_i$ par une association ta pour que la valeur $e2_j$.AT23 soit modifiée.

---

[17] On reprend dans chaque vue correspondant à un attribut dérivable TE.a, l'identifiant de TE : celui-ci permet d'associer une valeur de a à l'entité de TE à laquelle elle se rapporte.

[18] Une table complémentaire T' associée à T reprend l'identifiant de T : celui-ci permet d'associer les valeurs des attributs dérivables contenues dans T' à l'entité (tuple) correspondante dans T.

5.3 Système relationnel

On considère ici que les formules qui définissent les attributs dérivables sont des **relations** : les formules de définition établissent des relations entre les attributs d'un schéma. Cette approche peut être définie comme **non directionnelle** : les attributs résultats et les attributs données ne sont pas fixés à l'avance (on ne sait pas, avant l'exécution du schéma, quels sont les attributs pour lesquels les valeurs vont être fournies et quels sont les attributs pour lesquels il va falloir calculer les valeurs).

Les attributs dérivables ne sont plus nécessairement les attributs résultats : n'importe quel attribut (attribut de base ou attribut dérivable) peut, d'une exécution à l'autre du schéma, jouer le rôle d'attribut donnée ou d'attribut résultat. C'est au moment de l'exécution du schéma qu'on détermine quels sont les attributs pour lesquels on dispose des valeurs et qu'on calcule, dans la mesure du possible, les valeurs des attributs qu'on ne connaît pas.

L'exécution d'un schéma en système relationnel revient à résoudre un système de n équations (formules de définition) à m inconnues (attributs résultats) sur base des valeurs prises par les attributs données.

Reprenons l'exemple de la figure 5-1 et considérons à présent que :
- les attributs données sont `E1.AT12` et `E2.AT22`;
- les attributs résultats sont `E2.AT23` et `E3.AT32`.

Pour calculer les valeurs des attributs résultats, les formules de définition du schéma peuvent être réécrites sous la forme d'équations (ou relations) :

- En instanciant la formule de définition de `E1.AT12` à une entité quelconque $e1_i$ de `E1`, on obtient la formule :

  $e1_i$`.AT12 := E2(ta1:`$e1_i$`).AT23 * 4`

  Pour obtenir la valeur de l'attribut résultat `E2.AT23`, on peut transformer cette formule comme suit :

  `E2(ta1:`$e1_i$`).AT23 :=` $e1_i$`.AT12 / 4`

- En instanciant la formule de définition de `E2.AT23` à une entité quelconque $e2_i$ de `E2`, on obtient la formule :

  $e2_i$`.AT23 :=` $e2_i$`.AT22 + E3(ta2:`$e2_i$`).AT32`

  Pour obtenir la valeur de l'attribut résultat `E3.AT32`, on peut transformer cette formule comme suit :

  `E3(ta2:`$e2_i$`).AT32 :=` $e2_i$`.AT23 -` $e2_i$`.AT22`

Le tableau suivant donne, sur base d'un jeu de valeurs pour les attributs données, les valeurs correspondantes des attributs résultats :

Valeurs des attributs données	Attribut	Valeur des attributs résultats
96	$e1_1$.AT12	
48	$e1_2$.AT12	
60	$e1_3$.AT12	
15	$e2_1$.AT22	
13	$e2_2$.AT22	
19	$e2_3$.AT22	
	$e2_1$.AT23	24 ou 12
	$e2_2$.AT23	15
	$e2_3$.AT23	21
	$e3_1$.AT32	9 ou -3
	$e3_2$.AT32	2

Tableau 5-2 : Jeu de valeurs

On constate que $e2_1$.AT23 peut prendre deux valeurs distinctes : 24 ou 12. Ceci provient du fait que l'entité $e2_1$ est associée à $e1_1$ et $e1_2$ via ta1. On a donc :

E2(ta1:$e1_1$).AT23 := 96 / 4 et
E2(ta1:$e1_2$).AT23 := 48 / 4

Il existe donc deux valeurs de $e2_1$.AT23 qui vérifient les formules de définition. On peut faire la même remarque pour $e3_1$.AT32.

Si, pour une entité donnée, un attribut résultat peut, tout en respectant les formules de définition du schéma, prendre plusieurs valeurs, le mode d'exploitation en système relationnel fournit l'ensemble de toutes les valeurs admissibles.

Pratiquement, l'exploitation d'un schéma EA déductif en système relationnel peut se faire à l'aide de **résolveurs d'équations** [19]: le résolveur d'EXCEL ou TKSolver sont des logiciels parfaitement adaptés à ce mode d'exploitation pour de petits problèmes.

5.4 Optimisation

Dans ce dernier mode d'exploitation, le concepteur précise, au niveau du schéma EA déductif, une **fonction objectif** qu'il s'agit de **minimiser** ou de **maximiser**. Les valeurs des attributs résultats sont ensuite établies de sorte qu'elles **optimisent** (minimisent ou maximisent) les valeurs de la fonction objectif du schéma. Comme dans le mode d'exploitation précédent, les formules de définition sont vues comme des équations (ou relations) : attributs données et attributs résultats ne sont pas fixés à l'avance.

Associons, par exemple, au schéma de la figure 5-1 la fonction objectif suivante :

```
MAXIMISER E2.AT23
```

Il s'agit de déterminer les valeurs de tous les attributs résultats du schéma de manière à ce que les valeurs de E2.AT23 soient optimales (dans ce cas-ci, maximales).

---

[19] Un résolveur d'équations est un logiciel qui, à partir d'un système de n équations à m inconnues, fournit toutes les valeurs des inconnues qui résolvent le système d'équations, pour autant que celui-ci admette au moins une solution.

En reprenant le jeu de test du tableau 5-2, on choisira :

- parmi l'ensemble des valeurs admises pour $e2_1$.AT23, la valeur maximale, c'est-à-dire 24;
- parmi l'ensemble des valeurs admises pour $e3_1$.AT32, la valeur qui maximise $e2_1$.AT23 c'est-à-dire 9.

Pour les autres attributs, il n'existe qu'une seule solution possible.

Si, pour une entité donnée, un attribut résultat peut, tout en respectant les formules de définition du schéma, prendre plusieurs valeurs, le mode d'exploitation en optimisation détermine la valeur qui, parmi cet ensemble de valeurs admissibles, optimise la fonction objectif.

Pratiquement, l'exploitation d'un schéma EA déductif en optimisation peut se faire à l'aide d'outils de recherche opérationnelle se basant, par exemple, sur l'algorithme du Simplexe ([FICHEFET,82]) : OMP est un outil parfaitement adapté à ce mode d'exploitation.

5.5 Choix d'un mode d'exploitation

Les développements ultérieurs dépendent du mode d'exploitation particulier que nous choisissons d'approfondir. Dans ce travail, nous optons pour l'approche définitionnelle : les attributs données sont d'office les attributs de base tandis que les attributs résultats sont les attributs dérivables.

L'exécution d'un schéma revient donc à déterminer les valeurs des attributs dérivables à partir des valeurs des attributs de base et des formules de définition. Une formule de définition spécifie comment calculer les valeurs d'un attribut dérivable à partir des valeurs d'autres éléments du schéma. On se place dans une approche directionnelle où les formules sont vues comme des définitions et non comme des relations ou des équations.

6. *Analyse approfondie du modèle EA déductif*

Avant d'entamer l'étude de cohérence du modèle EA déductif, il est nécessaire d'introduire deux concepts qui s'avéreront particulièrement utiles dans la suite de ce travail :

- le premier est celui de **cardinalités d'une expression de désignation d'un attribut**;
- le second est celui de **graphe de dépendance d'un schéma EA déductif**.

6.1 Cardinalités d'une expression de désignation d'un attribut

Une expression de désignation d'un attribut (`Attr`) permet de désigner l'ensemble des valeurs prises par un certain nombre d'entités pour un attribut donné (cfr. *4.3.8*). Comme nous l'avons déjà signalé, une telle expression peut représenter 0,1 ou plusieurs valeurs.

Chaque expression de type `Attr` peut donc être caractérisée par le nombre minimum et le nombre maximum de valeurs que cette expression peut représenter. Comme pour les types d'associations, ces deux nombres `[i-j]` sont appelés cardinalités minimale (`i`) et maximale (`j`) de l'expression. Les cardinalités admises pour caractériser une expression `Attr` sont : `[1-1]`, `[0-1]`, `[0-N]` ou `[1-N]`.

L'objectif de cette section est d'analyser en détail une expression de désignation d'un attribut afin d'établir des règles permettant de déterminer les cardinalités d'une telle expression.

Considérons, par exemple, le schéma EA déductif de la figure 6-1 (les attributs dérivables sont indiqués en italique) :

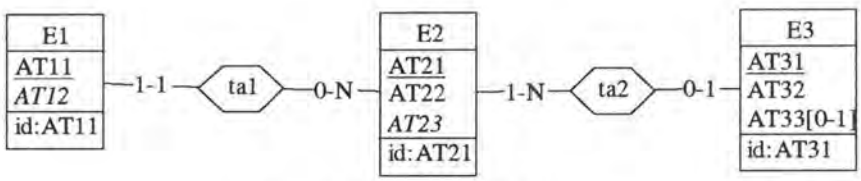


Figure 6-1 : Schéma EA déductif

Donnons maintenant quelques expressions `Attr` se rapportant à ce schéma et essayons intuitivement de déterminer leurs cardinalités :

- Dans la formule de définition de l'attribut dérivable `AT12` de `TE1`, l'expression `TE2(ta1:$).AT21` désigne avec certitude une et une seule valeur. En effet :
  - il n'y a qu'une et une seule entité de `TE2` associée à l'entité courante de `TE1`;
  - l'attribut `AT21` de `TE2` est obligatoire.

  On dit que les cardinalités de l'expression `TE2(ta1:$).AT21` sont `[1-1]`.

- Dans la formule de définition de l'attribut dérivable `AT23` de `TE2`, l'expression `TE3(ta2:$).AT33` désigne de 0 à N valeurs. En effet :
  - il peut y avoir de 1 à N entités de `TE3` associées à l'entité courante de `TE2`;
  - l'attribut `AT33` de `TE3` étant facultatif, il peut prendre 0 ou 1 valeur.

  On dit que les cardinalités de `TE3(ta2:$).AT33` sont `[0-N]`.

Comme nous venons de le voir, une expression de type `Attr` est caractérisée par des cardinalités. Il en sera de même pour d'autres constructions du langage : désignation d'un ensemble d'entités (`EnsEnt`), condition de sélection (`Csel`) et condition d'association (`Cass`). Donnons la signification des cardinalités pour chaque construction :

- les cardinalités d'une expression de désignation d'un ensemble d'entités `EnsEnt` représentent le nombre minimal et maximal d'entités que `EnsEnt` désigne;
- les cardinalités d'une condition de sélection `Csel` portant sur un type d'entités `TE` représentent le nombre minimal et maximal d'entités de `TE` que `Csel` sélectionne;
- les cardinalités d'une condition d'association `Cass` portant sur un type d'entités `TE` représentent le nombre minimal et maximal d'entités de `TE` que `Cass` sélectionne.

On désigne par **processus de calcul des cardinalités d'une expression Attr** le processus qui consiste à déterminer les cardinalités d'une expression de désignation d'un attribut. Ce processus se décompose en plusieurs sous-processus chargés chacun de déterminer les cardinalités d'une construction particulière du langage :

- Processus de calcul des cardinalités d'un ensemble d'entités (`EnsEnt`);
- Processus de calcul des cardinalités d'une condition de sélection (`Csel`);
- Processus de calcul des cardinalités d'une condition d'association (`Cass`).

Lorsque dans le calcul des cardinalités d'une construction, il n'est pas possible de déterminer en toute généralité la cardinalité minimale ou maximale de cette construction, on opte toujours pour la solution la plus générale c'est-à-dire la solution la moins restrictive :

- en cas d'incertitude sur la cardinalité minimale, c'est la cardinalité `0` qui l'emporte sur la cardinalité `1` : pour une cardinalité minimale, `0` est, en effet, moins restrictif que `1`;
- en cas d'incertitude sur la cardinalité maximale, c'est la cardinalité `N` qui l'emporte sur la cardinalité `1` : pour une cardinalité maximale, `N` est, en effet, moins restrictif que `1`.

6.1.1 Processus de calcul des cardinalités d'une expression de désignation d'un attribut

Une expression de type `Attr` se présente ainsi : `EnsEnt.Attribut`. Supposons que les entités de l'ensemble `EnsEnt` appartiennent au type d'entités `TE`.

Les cardinalités de l'expression `EnsEnt.Attribut` dépendent, d'une part, des cardinalités de `EnsEnt` et, d'autre part, du caractère obligatoire ou facultatif de l'attribut `TE.Attribut`. Les cardinalités de `EnsEnt` sont établies par le processus de calcul des cardinalités d'un ensemble d'entités (cfr. *6.1.2*).

Si `TE.Attribut` est obligatoire, les cardinalités de `EnsEnt.Attribut` sont entièrement déterminées par les cardinalités de `EnsEnt`.

Si par contre `TE.Attribut` est facultatif, alors cet attribut peut prendre la valeur `NULL` pour certaines entités de `TE`. Il est donc possible que `TE.Attribut` prenne la valeur `NULL` pour toutes les entités de `EnsEnt` et que, par conséquent, `EnsEnt.Attribut` représente 0 valeur. Donc, si `TE.Attribut` est facultatif, on fixe d'office la cardinalité minimale de `EnsEnt.Attribut` à 0.

Le tableau suivant résume le raisonnement en donnant les cardinalités de l'expression `EnsEnt.Attribut` en fonction des cardinalités de `EnsEnt` et du caractère obligatoire ou facultatif de `TE.Attribut`.

`EnsEnt.Attribut`		`TE.Attribut`	
		Obligatoire	Facultatif
Cardinalités de `EnsEnt`	`[1-1]`	`[1-1]`	`[0-1]`
	`[0-1]`	`[0-1]`	`[0-1]`
	`[1-N]`	`[1-N]`	`[0-N]`
	`[0-N]`	`[0-N]`	`[0-N]`

6.1.2 Processus de calcul des cardinalités d'un ensemble d'entités

Considérons que `E` est une expression de désignation d'un ensemble d'entités appartenant au type d'entités `TE`. En nous basant sur la définition du langage (cfr. *4.3*), envisageons les différentes formes possibles pour `E` :

1. `E` est de la forme `$`

Dans ce cas, `E` représente une entité unique. En effet, il n'y a, à un moment donné, qu'une et une seule entité courante. Les cardinalités de `E` sont `[1-1]`.

2. `E` est de la forme `I` (indice dans une fonction)

Dans ce cas, `E` représente une entité unique. Dans une itération donnée d'une fonction agrégative, l'indice `I` représente une et une seule entité. Les cardinalités de `E` sont `[1-1]`.

3. `E` est de la forme `TE`

Dans ce cas, `E` représente l'ensemble de toutes les entités appartenant au type d'entités `TE`. Comme il est impossible en toute généralité de déterminer le nombre d'entités d'un type d'entités au niveau de l'analyse conceptuelle, nous choisissons la solution la plus générale en fixant les cardinalités de `E` à `[0-N]`.

4. `E` est de la forme `TE(Csel)`

Dans ce cas, le nombre d'entités représentées par `E` dépend exclusivement des cardinalités de la condition de sélection `Csel`. Ces cardinalités sont déterminées par le processus de calcul des cardinalités d'une condition de sélection (cfr. *6.1.3*).

6.1.3 Processus de calcul des cardinalités d'une condition de sélection

En considérant que `Csel` est une condition de sélection portant sur un ensemble d'entités de type `TE`, et que $Csel_1$ et $Csel_2$ sont elles-mêmes des conditions de sélection, étudions les différentes formes que peut prendre `Csel` dans l'expression `TE(Csel)` :

1. `Csel` est de la forme $Csel_1$ `OR` $Csel_2$

`TE(`$Csel_1$ `OR` $Csel_2$`)` représente l'ensemble des entités de `TE` respectant la condition de sélection $Csel_1$ ou $Csel_2$. Chacune de ces deux conditions de sélection sélectionne un certain nombre d'entités : `TE(`$Csel_1$`)` et `TE(`$Csel_2$`)` désignent respectivement l'ensemble des entités de `TE` sélectionnées par $Csel_1$ et l'ensemble des entités sélectionnées par $Csel_2$.

L'ensemble d'entités TE(Csel$_1$ OR Csel$_2$) est, par conséquent, l'ensemble des entités appartenant à TE(Csel$_1$) ou à TE(Csel$_2$). L'ensemble TE(Csel$_1$ OR Csel$_2$) est donc constitué de l'union de TE(Csel$_1$) et de TE(Csel$_2$).

$$TE(Csel_1 \text{ OR } Csel_2) = TE(Csel_1) \cup TE(Csel_2)$$

La cardinalité maximale de Csel vaut toujours N : il est, en effet, impossible de savoir, au niveau de l'analyse conceptuelle, si l'union des deux ensembles TE(Csel$_1$) et TE(Csel$_2$) contient au plus une entité. Dans le doute, on prend la solution la plus générale qui consiste à fixer la cardinalité maximale à N.

La cardinalité minimale de Csel dépend de la cardinalité minimale de Csel$_1$ et Csel$_2$: il suffit qu'une des deux conditions de sélection Csel$_1$ ou Csel$_2$ sélectionne au minimum une entité pour être sûr d'avoir au moins une entité dans TE(Csel) (car union). Dans le cas où Csel$_1$ et Csel$_2$ sélectionnent toutes deux un minimum de 0 entité, il est impossible de déterminer en toute généralité si l'union des deux ensembles contiendra au minimum une entité : dans le doute, on fixe la cardinalité minimale de Csel à 0.

Le tableau suivant résume les différentes alternatives en donnant les cardinalités de l'expression Csel$_1$ OR Csel$_2$ en fonction des cardinalités de Csel$_1$ et de Csel$_2$.

Csel$_1$ OR Csel$_2$		Cardinalités de Csel$_1$			
		[1-1]	[0-1]	[1-N]	[0-N]
Cardinalités de Csel$_2$	[1-1]	[1-N]	[1-N]	[1-N]	[1-N]
	[0-1]	[1-N]	[0-N]	[1-N]	[0-N]
	[1-N]	[1-N]	[1-N]	[1-N]	[1-N]
	[0-N]	[1-N]	[0-N]	[1-N]	[0-N]

Pour déterminer les cardinalités de Csel$_1$ et de Csel$_2$, on applique récursivement le processus de calcul des cardinalités d'une condition de sélection sur Csel$_1$ et Csel$_2$.

2. Csel est de la forme Csel$_1$ AND Csel$_2$

TE(Csel$_1$ AND Csel$_2$) représente l'ensemble des entités de TE respectant la condition de sélection Csel$_1$ et Csel$_2$ c'est-à-dire l'ensemble des entités appartenant en même temps à TE(Csel$_1$) et à TE(Csel$_2$). L'ensemble TE(Csel$_1$ AND Csel$_2$) est donc constitué de l'intersection de TE(Csel$_1$) et de TE(Csel$_2$).

$$TE(Csel_1 \text{ AND } Csel_2) = TE(Csel_1) \cap TE(Csel_2)$$

La cardinalité minimale de Csel vaut toujours 0 : en effet, quelles que soient les cardinalités de Csel$_1$ et Csel$_2$, il est impossible de savoir, au niveau de l'analyse conceptuelle, si l'intersection des deux ensembles TE(Csel$_1$) et TE(Csel$_2$) contient au moins une entité. Dans le doute, on prend la solution la plus générale qui consiste à fixer la cardinalité minimale de Csel à 0.

La cardinalité maximale de Csel dépend de la cardinalité maximale de Csel$_1$ et Csel$_2$: il suffit qu'une des deux conditions de sélection Csel$_1$ ou Csel$_2$ sélectionne au maximum une entité pour être sûr d'avoir au maximum une entité dans TE(Csel) (car intersection). Dans le cas où Csel$_1$ et Csel$_2$ sélectionnent toutes deux un maximum de N entités, il est

impossible de déterminer en toute généralité si l'intersection des deux ensembles contient au maximum une entité : dans le doute, on fixe la cardinalité maximale de `Csel` à N.

Le tableau suivant résume les différentes alternatives en donnant les cardinalités de l'expression $Csel_1$ AND $Csel_2$ en fonction des cardinalités de $Csel_1$ et de $Csel_2$.

$Csel_1$ AND $Csel_2$		Cardinalités de $Csel_1$			
		[1-1]	[0-1]	[1-N]	[0-N]
Cardinalités de $Csel_2$	[1-1]	[0-1]	[0-1]	[0-1]	[0-1]
	[0-1]	[0-1]	[0-1]	[0-1]	[0-1]
	[1-N]	[0-1]	[0-1]	[0-N]	[0-N]
	[0-N]	[0-1]	[0-1]	[0-N]	[0-N]

Pour déterminer les cardinalités de $Csel_1$ et de $Csel_2$, on applique récursivement le processus de calcul des cardinalités d'une condition de sélection.

3. `Csel` est de la forme NOT $Csel_1$

TE(NOT $Csel_1$) représente l'ensemble des entités de `TE` ne respectant pas la condition de sélection $Csel_1$ c'est-à-dire l'ensemble des entités de `TE` n'appartenant pas à TE($Csel_1$). L'ensemble TE(NOT $Csel_1$) est donc constitué du complémentaire de l'ensemble TE($Csel_1$).

$$TE(\text{NOT } Csel_1) = \overline{TE(Csel_1)}$$

Même s'il est possible de déterminer les cardinalités de $Csel_1$, il est impossible, à partir de ces cardinalités, de déterminer en toute généralité les cardinalités de NOT $Csel_1$. Il faudrait, en effet, connaître pour cela le nombre total d'entités de `TE` au niveau de l'analyse conceptuelle. Par conséquent, dans le cas où `Csel` est constituée d'une négation, on fixe toujours ses cardinalités à [0-N] (cas le plus général).

4. `Csel` est constituée d'une condition d'association unique `Cass`

Dans ce cas, les cardinalités de `Csel` dépendent exclusivement des cardinalités de la condition d'association `Cass`. Ces cardinalités sont déterminées par le processus de calcul des cardinalités d'une condition d'association (cfr. *6.1.4*).

6.1.4 Processus de calcul des cardinalités d'une condition d'association

Supposons que `Cass` soit une condition d'association portant sur un ensemble d'entités du type d'entités `TE`. En nous basant sur la définition du langage (cfr. *4.3.11*), envisageons les différents cas possibles pour TE(Cass) :

1. `Cass` est une condition d'association avec des entités

Une condition d'association `Cass` avec les entités se présente ainsi : TAssociation:E_0. En remplaçant `Cass` par sa définition dans l'expression TE(Cass), on obtient l'écriture :

TE(TAssociation:E_0)

Supposons que les entités de E_0 appartiennent au type d'entités TE_0. La situation TE(TAssociation:E_0) se présente graphiquement comme indiqué à la figure 6-2.

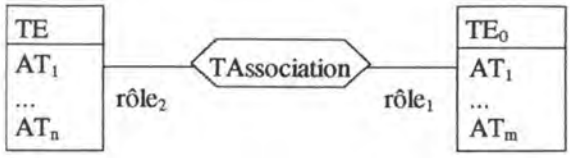


Figure 6-2 : Représentation graphique de TE(TAssociation:E0)

Les cardinalités de `TAssociation:E0` dépendent du nombre d'entités représentées par `E0` et des cardinalités du rôle `rôle1`. Il s'agit donc d'abord de déterminer les cardinalités de `E0` en appliquant récursivement le processus de calcul des cardinalités d'un ensemble d'entités sur `E0`.

Analysons une situation particulière où les cardinalités de `E0` valent `[1-1]` et où celles de `rôle1` valent `[0-N]`. L'ensemble d'entités `E0` contient donc une et une seule entité de `TE0`. Cette entité est associée à de "`0` à `N`" entités de `TE`. Le nombre d'entités de `TE` reliées via `TAssociation` à l'entité de `E0` est donc compris entre `0` et `N` : les cardinalités de `TAssociation:E0` sont `[0-N]`.

On pourrait tenir un raisonnement analogue pour toutes les autres combinaisons possibles de cardinalités. On peut énoncer la règle générale suivante :

- la cardinalité minimale de `TAssociation:E0` correspond à la plus petite des cardinalités minimales de `E0` et `rôle1`;
- la cardinalité maximale de l'expression correspond, elle, à la plus grande des cardinalités maximales de `E0` et `rôle1`.

Le tableau suivant résume les différentes alternatives en donnant les cardinalités de l'expression `TAssociation:E0` en fonction des cardinalités de `E0` et des cardinalités du rôle `rôle1`.

`TAssociation:E0`		Cardinalités de `E0`			
		`[1-1]`	`[0-1]`	`[1-N]`	`[0-N]`
Cardinalités du rôle `rôle1`	`[1-1]`	`[1-1]`	`[0-1]`	`[1-N]`	`[0-N]`
	`[0-1]`	`[0-1]`	`[0-1]`	`[0-N]`	`[0-N]`
	`[1-N]`	`[1-N]`	`[0-N]`	`[1-N]`	`[0-N]`
	`[0-N]`	`[0-N]`	`[0-N]`	`[0-N]`	`[0-N]`

2. `Cass` est une condition d'association avec des valeurs d'attribut

Une condition d'association `Cass` avec des valeurs d'attribut se présente ainsi : `:Attribut Rel EnsVal`. En remplaçant `Cass` par sa définition dans `TE(Cass)`, on obtient l'écriture :

```
TE(:Attribut Rel EnsVal)
```

A première vue, il est impossible de déterminer en toute généralité le nombre d'entités de `TE` respectant la condition posée sur `Attribut` puisque cette condition dépend de la valeur prise par `Attribut` pour chaque entité de `TE` et que cette valeur n'est pas connue au niveau de l'analyse conceptuelle. Dans l'incertitude, on fixe donc les cardinalités de l'expression `:Attribut Rel EnsVal` à `[0-N]` (cas le plus général).

Il existe cependant un cas particulier où on peut établir à coup sûr qu'il existe au maximum une entité de `TE` respectant la condition `:Attribut Rel EnsVal` : il s'agit du cas où

`Attribut` est l'identifiant de `TE` et où `Rel` est l'égalité (=). En effet, comme `Attribut` est l'identifiant de `TE`, on est certain qu'il n'existe pas deux entités de `TE` qui possèdent la même valeur pour `Attribut`. Dans ce cas particulier, la cardinalité maximale de `:Attribut Rel EnsVal` est donc `1`. La cardinalité minimale reste, quant à elle, à `0` : rien ne garantit, en effet, qu'il existe bien une entité de `TE` dont l'identifiant vaut `EnsVal`.

Les cardinalités de `:Attribut Rel EnsVal` sont donc déterminées comme suit :
- si `Attribut` est l'identifiant de `TE` et si `Rel` vaut =, alors les cardinalités sont `[0-1]`;
- sinon, les cardinalités sont `[0-N]`.

6.1.5 Algorithme de calcul des cardinalités

Nous donnons ici une synthèse du processus de calcul des cardinalités d'une expression de désignation d'un attribut sous la forme d'un algorithme. Dans les différentes fonctions, nous utilisons les conventions suivantes :

- si `Card` représente un couple de cardinalités, alors $Card_{min}$ représente la cardinalité minimale et $Card_{max}$ la cardinalité maximale;
- si i_1 et i_2 représentent deux valeurs entières, `Min(`i_1`,`i_2`)` représente le minimum de i_1, i_2 et `Max(`i_1`,`i_2`)` représente le maximum de i_1, i_2.

Un exemple complet d'application de la méthode sur des expressions de type `Attr` se trouve en annexe V.

Fonction `CARD_ATTR(Attr)`
Précondition : `Attr` doit être une expression de désignation d'un attribut. Elle doit donc pouvoir être décomposée en `EnsEnt.Attribut`.
Postcondition : `CARD_ATTR(Attr)` renvoie les cardinalités de `Attr`.

Algorithme :

```
Décomposer Attr en EnsEnt.Attribut
Card=CARD_ENSENT(EnsEnt)
Si Attribut est facultatif
      Alors Renvoyer    [0-Card_max]
      Sinon Renvoyer    Card
```

Fonction `CARD_ENSENT(E)`
Précondition : `E` doit être une expression de désignation d'un ensemble d'entités.
Postcondition : `CARD_ENSENT(E)` renvoie les cardinalités de `E`.

Algorithme :

```
Si E=$ ou E=I      Alors Renvoyer [1-1]
Si E=TE            Alors Renvoyer [0-N]
Si E=TE(Csel)      Alors Renvoyer CARD_CSEL(Csel,TE)
```

Fonction CARD_CSEL(Csel,TE)

Précondition : Csel doit être une condition de sélection et TE est le type d'entités sur lequel la condition Csel porte.

Postcondition : CARD_CSEL(Csel,TE) renvoie les cardinalités de Csel.

Algorithme :

```
Si Csel=Csel1 OR Csel2
      Alors  Card1:=CARD_CSEL(Csel1)
             Card2:=CARD_CSEL(Csel2)
             Si Card1min=1 ou Card2min=1
                    Alors Renvoyer [1-N]
                    Sinon Renvoyer [0-N]

Si Csel=Csel1 AND Csel2
      Alors  Card1:=CARD_CSEL(Csel1)
             Card2:=CARD_CSEL(Csel2)
             Si Card1max=1 ou Card2max=1
                    Alors  Renvoyer [0-1]
                    Sinon  Renvoyer [0-N]

Si Csel=NOT Csel1          Alors Renvoyer [0-N]

Si Csel=Cass               Alors Renvoyer CARD_CASS(Cass,TE)
```

Fonction CARD_CASS(Cass,TE)

Précondition : Cass doit être une condition d'association. TE doit être le type d'entités sur lequel la condition Cass porte.

Postcondition : CARD_CASS(Cass,TE) renvoie les cardinalités de Cass.

Algorithme :

```
Si Cass = :Attribut Rel EnsVal
      Alors  Si Attribut est identifiant de TE et Rel = '='
                    Alors Renvoyer [0-1]
                    Sinon Renvoyer [0-N]

Si Cass = TAssociation:E0     Les entités de E0 sont de type TE0
      Alors  Card1 := Cardinalités du rôle joué par TE0 dans TAssociation
             Card2 := CARD_ENSENT(E0)
             Renvoyer [Min(Card1min,Card2min) - Max(Card1max,Card2max)]
```

6.2 Graphe de dépendance des attributs

Le graphe de dépendance des attributs d'un schéma EA déductif a pour objectif de mettre en évidence les liens (dépendances) qui unissent les attributs de ce schéma. Il est établi à partir des formules de définition des attributs dérivables. Dans le graphe de dépendance :

- un attribut est représenté par un noeud du graphe;
- une dépendance est représentée par un arc.

On dira qu'un attribut `TE1.a` dépend d'un attribut `TE2.b` si la valeur de `b` de chaque entité de `TE2` doit être connue[20] pour pouvoir calculer la valeur de `a` de chaque entité de `TE1`. Plus précisément :

Si
- `TE1` et `TE2` sont deux types d'entités du schéma EA déductif,
- $\{e1_1, \ldots, e1_n\}$ sont les entités de `TE1`,
- $\{e2_1, \ldots, e2_m\}$ sont les entités de `TE2`,
- `a` est un attribut de `TE1`,
- `b` est un attribut de `TE2`,

alors `TE1.a` dépend de `TE2.b` ssi les valeurs de $e2_1.b, \ldots, e2_m.b$ doivent être connues avant de pouvoir établir les valeurs de $e1_1.a, \ldots, e1_m.a$.

Remarque : En fait, les valeurs prises par les entités de `TE2` pour l'attribut `b` ne doivent pas nécessairement être toutes connues pour pouvoir calculer les valeurs de `TE1.a`. Nous ferons cependant l'hypothèse simplificatrice suivante : si la valeur de `b` d'au moins une entité de `TE2` doit être connue, alors la valeur de `b` de toutes les entités de `TE2` doit être connue.

On distingue deux types de dépendances :

1. **dépendance directe** : l'attribut `TE1.a` dépend directement de l'attribut `TE2.b` si `TE2.b` intervient explicitement dans la formule de définition de `TE1.a`. En effet, si on fait référence à l'attribut `TE2.b` dans la formule de définition de `TE1.a`, c'est que les valeurs de `TE2.b` doivent être connues pour pouvoir établir les valeurs de `TE1.a`. Dans le graphe de dépendance, ce fait sera représenté par un arc orienté partant de `TE2.b` vers `TE1.a`.

   `TE2.b` ⟶ `TE1.a`

2. **dépendance indirecte** : l'attribut `TE1.a` dépend indirectement de l'attribut `TE3.c` si `TE1.a` dépend directement d'un attribut `TE2.b` qui dépend lui-même (directement ou indirectement) de `TE3.c`. Le fait que `TE1.a` dépende indirectement de `TE3.c` n'est pas indiqué explicitement dans le graphe mais cette dépendance peut être déduite de la présence d'un chemin reliant `TE3.c` à `TE1.a`.

   `TE3.c` ⟶ `TE2.b` ⟶ `TE1.a`

[20] Dans le cas où `TE2.b` est facultatif, la valeur NULL est considérée ici comme une valeur classique, au même titre que les autres : s'il est établi qu'une entité de `TE2` prend la valeur NULL pour l'attribut `b`, cette valeur est considérée connue.

L'objectif du graphe de dépendance des attributs est donc de déterminer, pour chaque attribut dérivable `TE.a`, tous les attributs dont dépend `TE.a`. Le graphe est construit en deux étapes :

1. pour chaque formule de définition, on construit le **graphe de dépendance local à cette formule**;
2. on regroupe ensuite les graphes de dépendance locaux et on construit le **graphe de dépendance complet du schéma**.

6.2.1 Construction du graphe local à une formule

On analyse d'abord individuellement chaque formule de définition et on construit le graphe local à cette formule. L'objectif du graphe local à une formule est de mettre en évidence les dépendances directes.

Un attribut dérivable dépend directement de chaque attribut figurant dans sa formule de définition. Passons rapidement en revue les différentes possibilités de désigner un attribut dans une formule.

6.2.1.1 Expression de désignation d'un attribut (Attr)

Il s'agit du cas où on fait référence à un attribut via une expression `Attr`. Supposons que l'attribut dérivable `a` du type d'entités `TE` contienne dans sa formule de définition une expression `Attr` (à n'importe quel endroit où une telle expression est admise).

L'expression `Attr` se présente ainsi : `EnsEnt.Attribut`. Sur base de la définition du langage, analysons les différentes formes possibles pour `EnsEnt` (cfr. *4.3.9*) :

- Si `EnsEnt=TEntité` où `TEntité` est un type d'entités du schéma,
  alors l'attribut dérivable `TE.a` dépend directement de `TEntité.Attribut`.
- Si `EnsEnt=$` où `$` représente l'entité courante de type `TE`,
  alors l'attribut dérivable `TE.a` dépend directement de `TE.Attribut`.
- Si `EnsEnt=I` où `I` est un indice représentant une entité de type `TEntité`,
  alors l'attribut dérivable `TE.a` dépend directement de `TEntité.Attribut`.
- Si `EnsEnt=TEntité(Csel)` où `TEntité` est un type d'entités du schéma et `Csel` une condition de sélection portant sur `TEntité`,
  alors l'attribut dérivable `TE.a` dépend directement de `TEntité.Attribut`

6.2.1.2 Condition d'association avec des valeurs d'attribut

Il s'agit du cas où on fait référence à l'attribut `Attribut` via une condition d'association avec des valeurs d'attribut `TEntité(:Attribut Rel Ensval)`. L'attribut dérivable défini par une formule de définition contenant une telle expression dépend directement de `TEntité.Attribut`.

6.2.1.3 Exemple

Nous basons nos exemples sur le schéma EA déductif de la figure 6-3 (les attributs dérivables sont indiqués en italique).

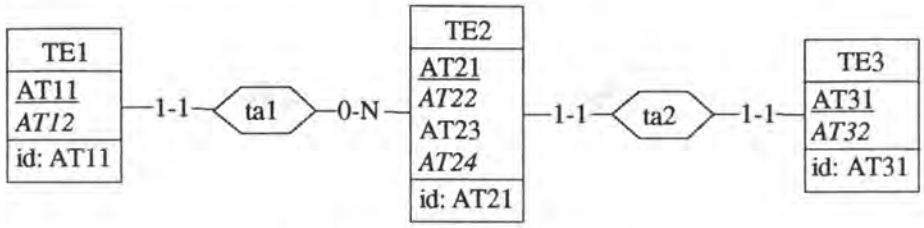
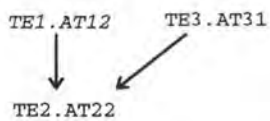


Figure 6-3 : Schéma EA déductif

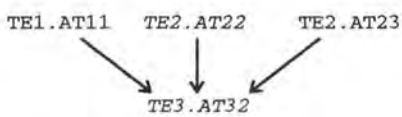
Voici les formules de définition des attributs dérivables ainsi que les graphes de dépendance locaux à chaque formule :

- `TE1.AT12` est défini par `DEF= $.AT11 + TE2(ta1:$).AT21 * TE3(ta2:TE2(ta1:$)).AT31`

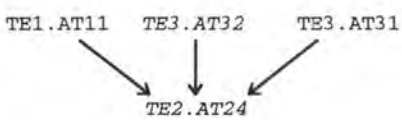
- `TE2.AT22` est défini par `DEF= Min(I=TE1;I.AT12) * TE3(ta2:$).AT31`



- `TE3.AT32` est défini par `DEF= Moyenne(I=TE2(:AT23 in TE1.AT11);I.AT23) / TE2(ta2:$).AT22`



- `TE2.AT24` est défini par `DEF= Nombre(TE1(ta1:$ AND :AT11<5)) + TE3(ta2:$).AT32 * TE3(ta2:$).AT31`



6.2.2 Construction du graphe de dépendance complet du schéma

L'objectif du graphe de dépendance complet est d'avoir une vue globale des dépendances : on retrouve non seulement les dépendances directes mais aussi les dépendances indirectes.

Pour obtenir le graphe de dépendance complet du schéma, on procède comme suit :

1. on crée un noeud dans le graphe complet pour chaque attribut (dérivable ou non) du schéma EA déductif;
2. on reproduit, dans le graphe complet, toutes les dépendances directes (symbolisées par des arcs orientés) mises en évidence dans les différents graphes locaux.

Le graphe de dépendance complet du schéma de la figure 6-3 est le suivant :

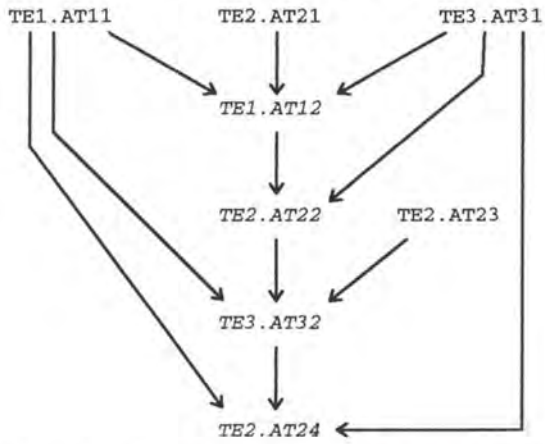


Figure 6-4 : Graphe de dépendance complet du schéma

Remarquons que dans le graphe de dépendance complet d'un schéma :

- un attribut dérivable peut être destination d'un nombre quelconque d'arcs orientés, y compris aucun (au cas où il ne dépend d'aucun autre attribut);
- un attribut dérivable peut être origine d'un nombre quelconque d'arcs orientés. Un attribut dérivable peut, en effet, intervenir dans la définition d'autres attributs dérivables;
- un attribut de base peut être origine d'un nombre quelconque d'arcs orientés;
- un attribut de base ne peut être destination d'aucun arc orienté. En effet, un attribut de base ne dépend d'aucun autre attribut (sinon, il serait attribut dérivable et non attribut de base);
- s'il existe un arc orienté partant d'un attribut `a` vers un attribut `b`, `b` dépend directement de `a`. S'il existe un chemin de longueur supérieure à 1 dont l'origine est un attribut `a` et la destination un attribut `b`, `b` dépend indirectement de `a`;
- s'il existe un chemin de longueur quelconque (supérieure à 0) dont l'origine est un attribut `a` et la destination un attribut `b`, `b` dépend de `a`.

On appelle **graphe de dépendance réduit du schéma**, le graphe de dépendance du schéma dans lequel ne figurent que les attributs dérivables : on l'obtient en éliminant tous les attributs de base dans le graphe complet.

Le graphe de dépendance réduit correspondant au schéma de la figure 6-3 s'obtient à partir du graphe de dépendance complet de la figure 6-4 et se présente ainsi :

Figure 6-5 : Graphe de dépendance réduit du schéma

Le graphe de dépendance réduit d'un schéma EA déductif sera particulièrement utile lors de l'étude de cohérence de ce schéma. Il permettra notamment de repérer les définitions récursives. De plus, le graphe réduit est suffisamment riche pour satisfaire à toutes les analyses de cohérence : nous ne nous embarrasserons donc plus du graphe de dépendance complet, beaucoup plus dense et beaucoup plus lourd à manipuler.

Le graphe de dépendance complet n'est cependant pas totalement inutile : il permet au concepteur de bien percevoir la structure globale d'un schéma EA déductif en visualisant tous les attributs et les relations de dépendance qui les unissent.

7. Etude de cohérence d'un schéma EA déductif

En toute généralité, un schéma EA déductif est correct s'il se comporte comme le domaine d'application qu'il est chargé de décrire ([HAINAUT,94a]). S'il est le plus souvent impossible de vérifier de manière absolue qu'un schéma jouit de cette propriété, on peut néanmoins énoncer quelques règles que tout schéma doit respecter. Un schéma qui respecte ces règles peut être correct (sans que cela soit garanti), tandis qu'un schéma qui ne les respecte pas est soit incorrect, soit susceptible de produire, tôt au tard, des résultats erronés.

L'objectif de l'étude de cohérence est d'avertir le concepteur d'un schéma EA déductif de la présence éventuelle d'incohérences dans son schéma. Nous distinguons deux types d'incohérences en fonction du degré de certitude avec lequel ces incohérences peuvent être détectées :

- **incohérences certaines** : incohérences en présence desquelles il est impossible que le schéma fournisse des résultats.

  Exemple : Une formule de définition contient un attribut qui n'existe pas.

- **incohérences possibles** : incohérences en présence desquelles le schéma est susceptible de fournir à un moment donné des résultats erronés.

  Exemple : Une expression simple dans une expression arithmétique est susceptible de désigner plus d'une valeur.

Dans ce chapitre, nous énonçons un certain nombre de **règles de cohérence**. Nous distinguons deux types de cohérence : la **cohérence syntaxique** et la **cohérence sémantique**.

7.1 Cohérence syntaxique

Pour qu'un schéma EA déductif soit cohérent au niveau syntaxique, il faut que toutes les formules de définition de ses attributs dérivables soient syntaxiquement correctes. Une formule de définition est syntaxiquement correcte si elle respecte la syntaxe du langage d'expression des formules de définition (cfr. *4.3*).

Toute incohérence au niveau syntaxique doit être considérée comme une incohérence certaine. Voici, par exemple, quelques règles de définition syntaxiquement incorrectes :

```
DEF= Min(LIGNECOM.QCOM)

DEF= FOURNISSEUR.3PRIX

DEF= $ - ($.PRIX_VENTE / 2

DEF= Somme( I=COMMANDE;
            Somme(I=LIGNECOM(concerne:I);I.QCOM)-I.MONTANT_PAYE)
```

7.2 Cohérence sémantique

On distingue deux catégories de cohérence sémantique :

- **cohérence sémantique locale** : on analyse individuellement la formule de définition de chaque attribut dérivable;
- **cohérence sémantique globale** : on analyse les relations entre les formules de définition des attributs dérivables d'un schéma en se basant sur son graphe de dépendance réduit.

7.2.1 Cohérence sémantique locale

7.2.1.1 Cohérence des types

L'objectif est de déterminer si une formule de définition est cohérente au niveau des types de valeurs qu'elle met en relation. Pour réaliser cette analyse, on se base sur les règles établies en *4.3.2.*

Il faut, d'une part, que le type de l'expression de la formule de définition d'un attribut dérivable `a` soit le même que le type de `a` : si `a` a comme formule de définition `DEF=expr`, il faut que `a` et `expr` soient de même type.

D'autre part, il faut que, au sein même d'une expression, les opérateurs utilisés mettent en correspondance des valeurs dont les types soient compatibles entre eux et avec l'opérateur.

Exemples :

- Si $expr_1$ est une expression alphanumérique et $expr_2$ une expression numérique, on ne peut pas écrire $expr_1 = expr_2$.
- Si $expr_1$ et $expr_2$ sont deux expressions booléennes, on ne peut pas écrire $expr_1 < expr_2$.
- Si $expr_1$ et $expr_2$ sont deux expressions alphanumériques, on ne peut pas écrire $expr_1 + expr_2$.

Toute incohérence au niveau des types doit être considérée comme incohérence certaine.

En considérant que `expr` est une expression figurant dans une formule de définition (à n'importe quel endroit où une expression est admise), envisageons les différents cas possibles :

1. Si `expr` est une expression arithmétique d'une des deux formes
   - $expr_1$
   $expr_2$ ω $expr_3$
   où ω est l'un des opérateurs binaires `+`,`-`,`*` ou `/`,

   alors $expr_1$, $expr_2$ et $expr_3$ doivent être de type numérique.

2. Si `expr` est une expression booléenne d'une des deux formes
   `NOT` $expr_1$
   $expr_2$ ω $expr_3$
   où ω est l'un des opérateurs logiques `AND` ou `OR`,

   alors $expr_1$, $expr_2$ et $expr_3$ doivent être de type booléen.

3. Si `expr` est une formule atomique de la forme
   $expr_1$ ω $expr_2$
   où ω est l'un des opérateurs de comparaison <,>,<=,>=,= ou <>,

   alors $expr_1$ et $expr_2$ doivent être de même type : numérique, alphanumérique ou booléen. De plus, si $expr_1$ et $expr_2$ sont de type booléen, il faut que ω soit = ou <>.

Un contrôle de type est également nécessaire dans les appels de fonction :

1. Si on a un appel de fonction d'une des formes
   `Min(I=E;`$expr_I$`)`
   `Max(I=E;`$expr_I$`)`
   où - `E` désigne un ensemble d'entités;
   - $expr_I$ désigne une expression qui dépend de l'indice `I`,

   alors $expr_I$ doit être de type numérique ou alphanumérique.

2. Si on a un appel de fonction de la forme
   `Somme(I=E;`$expr_I$`)`
   `Moyenne(I=E;`$expr_I$`)`
   où - `E` désigne un ensemble d'entités;
   - $expr_I$ désigne une expression qui dépend de l'indice `I`,

   alors $expr_I$ doit être de type numérique.

Les fonctions `Nombre` et `NombreVal` ne sont soumises à aucune contrainte de type.

Le dernier contrôle de type porte sur les conditions d'association avec des valeurs d'attribut :

1. Si on a une condition d'association avec des valeurs d'attribut de la forme
   `TEntité(:Attribut Rel EnsVal)`
   où - `EnsVal` est une expression simple `expr`;
   - `Rel` vaut <,>,<=,>=,=,<>, `in`, `not in`;
   - `TEntité` est un type d'entités du schéma EA déductif;
   - `Attribut` est un attribut de `TEntité`,

   alors `TEntite.Attribut` et la (les) valeur(s) représentée(s) par `expr` doivent être du même type. De plus, si `TEntite.Attribut` et `expr` sont de type booléen, il faut que `Rel` soit obligatoirement <>, =, `in` ou `not in`.

2. Si on a une condition d'association avec des valeurs d'attribut de la forme
   `TEntité(:Attribut Rel EnsVal)`
   où - `EnsVal` est constituée d'un ensemble d'expressions simples
   {$expr_1$, . . . , $expr_n$};
   - `Rel` vaut `in`, `not in`;
   - `TEntité` est un type d'entités du schéma EA déductif;
   - `Attribut` est un attribut de `TEntité`;

   alors - les valeurs désignées par {$expr_1$, . . . , $expr_n$} doivent être du même type;
   - `TEntite.Attribut` et les valeurs désignées par {$expr_1$, . . . , $expr_n$} doivent être du même type.

7.2.1.2 Cohérence des noms

Il faut que tous les noms des attributs, des types d'associations et des types d'entités qui figurent dans les formules de définition des attributs dérivables existent dans le schéma EA déductif et qu'ils soient correctement utilisés. Plus précisément :

1. Dans une expression telle que
   `TEntité(TAssociation:EnsEnt)`
   où `EnsEnt` désigne un ensemble d'entités appartenant à un même type d'entités `TE`,

   il faut que
   - `TEntité` soit le nom d'un type d'entités du schéma EA déductif;
   - `TAssociation` soit le nom d'un type d'associations reliant `TEntité` à `TE`.

   Au cas où `TAssociation` est un type d'associations récursif, `TEntité` peut désigner le nom d'un rôle joué par un type d'entités dans `TAssociation`.

2. Dans une expression telle que
   `TEntité(:Attribut rel EnsVal)`

   il faut que
   - `TEntité` soit le nom d'un type d'entités du schéma EA déductif;
   - `Attribut` soit le nom d'un attribut de `TEntité`.

3. Dans une expression telle que
   `EnsEnt.Attribut`
   où `EnsEnt` désigne un ensemble d'entités appartenant à un même type d'entités `TE`,

   il faut que `Attribut` soit le nom d'un attribut de `TE`.

4. Dans une expression `TEntité` désignant toutes les entités d'un type, il faut que `TEntité` soit le nom d'un type d'entités du schéma EA déductif.

Toute incohérence au niveau du nom des attributs, des types d'associations ou des types d'entités doit être considérée comme une incohérence certaine.

Exemple : Considérons le schéma EA déductif suivant :

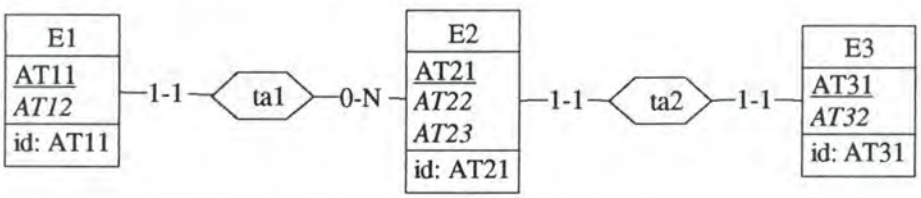


Figure 7-1 : Schéma EA déductif

En se basant sur ce schéma, voici une formule de définition incorrecte de `E1.AT12` :

```
DEF= $.AT21 +                          (1)
     E2(ta2:$).AT21 *                  (2)
     E3(ta3:H(ta1:$)).AT11 /           (3)
     Min(I=E1(:F < 120);I.AT11) +      (4)
     Nombre(G)                         (5)
```

Cette définition est incorrecte parce que :
- Ligne (1) : `AT21` n'est pas un attribut de `E1`;
- Ligne (2) : `ta2` n'est pas un type d'associations reliant `E2` à `E1`;
- Ligne (3) : `ta3` n'est pas un type d'associations;
  `H` n'est pas un type d'entités du schéma;
  `AT11` n'est pas un attribut de `E3`;
- Ligne (4) : `F` n'est pas un attribut de `E1`;
- Ligne (5) : `G` n'est pas un type d'entités du schéma.

7.2.1.3 Evaluation d'une expression de type `Attr`

Nous avons vu qu'une expression de type `Attr` est la seule expression simple pouvant représenter 0, 1 ou plusieurs valeurs[21]. Nous avons également élaboré une méthode qui calcule les cardinalités minimale et maximale d'une telle expression. Dans certaines situations, `Attr` doit représenter au maximum une valeur, dans d'autres elle peut représenter plusieurs valeurs. On s'intéresse ici uniquement à la cardinalité maximale d'une expression `Attr`.

Le processus de calcul des cardinalités d'une expression `Attr` ne fournit pas un résultat certain dans tous les cas : en effet, nous avons vu qu'en cas de doute dans le calcul des cardinalités, la méthode optait toujours pour la solution la plus générale. Il est par conséquent possible que, pour une expression `Attr` donnée, la méthode nous indique une cardinalité maximale de `N` alors qu'elle vaut en réalité `1` (pour les cardinalités maximales, `N` est plus général que `1`).

Exemple : En se basant sur le schéma de la figure 4-1 et en considérant que, dans le domaine d'application modélisé, il existe au maximum un client par localité, l'expression `Attr CLIENT(:LOCALITE="NAMUR").COMPTE` désigne avec certitude une valeur au maximum. En analysant cette expression, le processus de calcul des cardinalités déduira cependant une cardinalité maximale de `N`.

Les incohérences détectées au niveau de l'évaluation d'une expression `Attr` doivent donc être considérées comme des incohérences possibles : c'est au concepteur de déterminer s'il y a effectivement erreur ou s'il s'agit d'une fausse alerte.

Nous allons maintenant passer en revue les différentes situations où une expression de type `Attr` peut être utilisée et préciser, pour chacune de celles-ci, si `Attr` doit représenter au maximum une valeur ou si `Attr` peut en représenter plusieurs.

---

[21] Les autres types d'expressions simples (appel de fonction et constante) représentent toujours 0 ou 1 valeur.

1. `Attr` utilisé dans une expression arithmétique ou logique[22]

Dans ce cas, il faut obligatoirement que la cardinalité maximale de `Attr` soit 1.

Exemple (basé sur le schéma de la figure 7-1) :

- Définition de `E1.AT12` :
  ```
  DEF=  $.AT11 + E2(ta1:$).AT23 * E3(ta2:E2(ta1:$)).AT31
  ```
  Chaque expression `Attr` figurant dans cette formule de définition représente au maximum une seule valeur : pas de problème.

- Définition de `E2.AT22` :
  ```
  DEF=    E3.AT31
  ```
  L'expression `Attr` qui figure dans cette formule de définition représente plusieurs valeurs d'attribut (l'ensemble des valeurs prises par toutes les entités de `E3` pour l'attribut `AT31`): elle n'est pas admise ici.

- Définition de `E2.AT23` :
  ```
  DEF= Min(I=E2 ; I.AT21 * E1(ta1:I).AT11)
  ```
  L'expression `I.AT21` qui figure dans l'expression arithmétique (deuxième argument de la fonction `Min`) représente une valeur au maximum mais l'expression `E1(ta1:I).AT11` est susceptible de représenter plusieurs valeurs : elle n'est pas admise ici.

2. `Attr` utilisé dans une condition d'association avec des valeurs d'attribut

On distingue deux cas :

- Si on a une condition d'association avec des valeurs d'attribut de la forme
  `TEntité(:Attribut Rel Attr)`
  où
  - `TEntité` est un type d'entités du schéma EA déductif;
  - `Attribut` est un attribut de `TEntité`;

  alors :
  - si `Rel` est un opérateur de comparaison <,>,<=,>=,= ou <>, alors la cardinalité maximale de `Attr` doit être `1`;
  - si `Rel` est un opérateur ensembliste `in` ou `not in`, alors la cardinalité maximale de `Attr` peut être `N`.

- Si on a une condition d'association avec des valeurs d'attribut de la forme
  `TEntité(:Attribut Rel {`$expr_1, \ldots, expr_n$`})`
  où
  - `TEntité` est un type d'entités du schéma EA déductif;
  - `Attribut` est un attribut de `TEntité`;
  - $expr_1, \ldots, expr_n$ sont des expressions simples;

  alors toute expression $expr_i$ ($1 \le i \le n$) étant de type `Attr` peut avoir une cardinalité maximale de `N`.

---

[22] Ce cas englobe la possibilité où la formule de définition d'un attribut dérivable est constituée d'une seule expression simple de type `Attr`.

7.2.1.4 Valeur NULL et attribut facultatif

Si l'expression `expr` qui constitue la formule de définition d'un attribut dérivable `a` de `TE` est susceptible de prendre la valeur `NULL` (0 valeur) pour certaines entités de `TE`, il faut que `TE.a` soit déclaré facultatif.

`expr` est constitué d'expressions simples reliées entre elles par des opérateurs. Il suffit qu'une seule de ces expressions simples prenne la valeur `NULL` pour que `expr` toute entière prenne la valeur `NULL`. Analysons individuellement chaque type d'expressions simples en déterminant les circonstances dans lesquelles elles peuvent prendre la valeur `NULL` :

1. Une constante représente toujours une et une seule valeur et ne prend donc jamais la valeur `NULL`;
2. Une expression de type `Attr` peut représenter la valeur `NULL`. Nous disposons d'une méthode capable de déterminer les cardinalités d'une expression `Attr` (cfr. *6.1.1*). Si la cardinalité minimale de `Attr` vaut 0, cela signifie que `Attr` est susceptible de prendre la valeur `NULL`. Si par contre elle vaut 1, c'est que `Attr` ne prendra jamais la valeur `NULL`;
3. Les appels de fonction `Min`, `Max`, `Moyenne` et `Somme` peuvent, selon les cardinalités de leurs arguments, prendre la valeur `NULL` (cfr. *4.4*). Les appels de fonction `Nombre` et `NombreVal` représentent toujours une et une seule valeur.

Pour déterminer si les appels de fonction `Somme(I=E;`$\texttt{expr}_I$`)`, `Min(I=E;`$\texttt{expr}_I$`)`, `Max(I=E;`$\texttt{expr}_I$`)` ou `Moyenne(I=E;`$\texttt{expr}_I$`)` sont susceptibles de prendre la valeur `NULL`, on utilise le processus de calcul des cardinalités d'un ensemble d'entités (cfr. *6.1.2*) : si la cardinalité minimale de `E` vaut 0, cela signifie que `E` est susceptible de désigner un ensemble de 0 entité au quel cas l'appel de fonction peut prendre la valeur `NULL`. Si par contre elle vaut 1, c'est que `E` désigne toujours un minimum d'une entité.

Il est nécessaire aussi de déterminer si l'expression $\texttt{expr}_I$ est susceptible de prendre la valeur `NULL` pour certaines entités de `E`. En effet, si $\texttt{expr}_I$ peut prendre la valeur `NULL` pour certaines entités de `E`, c'est que $\texttt{expr}_I$ peut prendre la valeur `NULL` pour chaque entité de `E` au quel cas l'appel de fonction tout entier prend la valeur `NULL`.

$\texttt{expr}_I$ est constituée d'une ou de plusieurs expressions simples reliées entre elles par des opérateurs. Ces expressions simples sont soit des expressions de désignation d'un attribut, soit des constantes. Comme les constantes ne prennent jamais la valeur `NULL`, il suffit de déterminer, pour chaque expression de type `Attr` de $\texttt{expr}_I$, si elle peut prendre la valeur `NULL`. On utilise, à cette fin, le processus de calcul des cardinalités d'une expression de type `Attr` (cfr. *6.1.1*) :

- si une expression de type `Attr` de $\texttt{expr}_I$ peut prendre la valeur `NULL`, c'est que l'expression $\texttt{expr}_I$ peut prendre la valeur `NULL` pour certaines entités de `E` et donc que l'appel de fonction tout entier est susceptible de prendre la valeur `NULL`;
- sinon, pour toute entité de `E`, $\texttt{expr}_I$ ne prend jamais la valeur `NULL`.

C'est seulement si `E` désigne avec certitude au moins une entité et si $\texttt{expr}_I$ ne prend jamais la valeur `NULL` que l'appel de fonction tout entier ne prend jamais la valeur `NULL`. Dans les autres cas, l'appel de fonction est susceptible de désigner la valeur `NULL`.

De nouveau, les méthodes mises au point pour déterminer les cardinalités d'une expression `Attr` et d'une expression `EnsEnt` (expression de désignation d'un ensemble d'entités) ne fournissent pas un résultat certain dans tous les cas : en cas de doute, les méthodes optent toujours pour la solution la plus générale. Il est par conséquent possible que, pour une expression `Attr` ou `EnsEnt` donnée, les méthodes nous indiquent une cardinalité minimale de `0` alors qu'elle vaut en réalité `1` (pour les cardinalités minimales, `0` est plus général que `1`).

Exemple : En se basant sur la figure 4-1 et en considérant que, dans le domaine d'application modélisé, les clients appartiennent à la catégorie 1 ou 2, l'expression `EnsEnt CLIENT(:CATEGORIE in {1,2})` désigne avec certitude au moins une entité [23]. En analysant cette expression, le processus de calcul déduira cependant une cardinalité minimale de `0`.

Toute incohérence détectée dans cette partie de l'analyse sémantique doit donc être considérée comme une incohérence possible : c'est au concepteur de déterminer s'il y a effectivement erreur ou non.

7.2.1.5 Cohérence des unités

On n'additionne pas, dit-on, des pommes et des poires. Il en est de même pour les expressions qui apparaissent dans un schéma EA déductif.

Exemple :

- on ne peut pas additionner des kilogrammes avec des grammes, des mètres avec des centimètres, des francs avec des centimes, etc.;
- cela n'a pas de sens d'additionner des grammes et des mètres par seconde (vitesse), des kilomètres et des francs, etc.;
- cela n'a pas de sens d'additionner des Francs belges avec des Lires, des Marks avec des Florins, etc.

Au sein d'une formule de définition, il faut que les expressions simples mises en correspondance aient des unités cohérentes. Le modèle EA déductif tel que nous l'avons défini ne connaît pas le concept d'unité : pour pouvoir effectuer un contrôle systématique des unités mises en correspondance, il faudrait définir, pour chaque attribut dérivable du schéma, l'unité dans laquelle cet attribut s'exprime (Kg, g, m, Km, m/s, BEF, ...). Il serait intéressant aussi de disposer d'une table de correspondance entre unités : on pourrait, par exemple, y exprimer que 1 mètre = 100 centimètres. Il serait alors possible de transformer les formules de définition pour que toutes les expressions simples la constituant s'expriment dans une même unité : on ramènerait, par exemple, les centimètres en mètres en les divisant par 100.

Dans le schéma EA déductif, il n'est donc pas possible d'effectuer une analyse systématique des unités : c'est au concepteur de s'assurer qu'il n'y a pas d'incohérence au niveau des unités mises en correspondance dans son schéma.

---

[23] En faisant l'hypothèse réaliste qu'il existe au moins une entité `CLIENT`.

7.2.2 Cohérence sémantique globale

7.2.2.1 Formules de définition récursives

On dit que les formules de définition des attributs dérivables $a_1, \ldots, a_n$ d'un schéma EA déductif sont récursives lorsque chaque attribut a_i ($1 \leq i \leq n$) dépend de lui-même.

Comme nous l'avons déjà défini (cfr. *6.2*), un attribut dérivable `a` dépend d'un attribut `b` lorsque les valeurs de `b` interviennent dans l'établissement des valeurs de `a`. Si `a` dépend de lui-même, cela signifie donc qu'il faudrait connaître les valeurs de `a` pour les calculer, ce qui est absurde. Par conséquent, les définitions récursives ne peuvent pas être admises : un attribut dérivable ne peut pas dépendre de lui-même.

La présence de définitions récursives au sein d'un graphe EA déductif rend impossible le calcul des valeurs de certains attributs dérivables : elle doit donc être considérée comme une incohérence certaine.

Pour détecter la présence de formules de définition récursives, on utilise le graphe de dépendance des attributs (cfr. *6.2*). La présence d'un circuit dans le graphe de dépendance indique l'existence de formules de définition récursives : si les attributs dérivables $a_1, \ldots, a_n$ constituent un circuit dans le graphe de dépendance, les formules de définition de ces attributs sont récursives.

En effet, si dans le graphe a_i fait partie d'un circuit, cela signifie qu'il existe un chemin de longueur supérieure à 0 reliant a_i à lui-même et, donc, que a_i dépend de lui-même.

Comme, d'une part, on utilise le graphe de dépendance uniquement pour détecter la présence de circuits et que, d'autre part, il est impossible que les attributs de base fassent partie d'un tel circuit[24], on peut se contenter de travailler sur le graphe de dépendance réduit du schéma.

Considérons, par exemple, les formules de définition suivantes se rapportant au schéma EA déductif de la figure 7-1 :

- `E1.AT12` est défini par
```
DEF= $.AT11 +
     E2(ta1:$).AT23 * E3(ta2:E2(ta1:$)).AT31
```
- `E2.AT22` est défini par
```
DEF= Min(I=E1;I.AT12)
```
- `E3.At32` est défini par
```
DEF= Moyenne(I=E2;I.AT21) / E2(ta2:$).AT22
```
- `E2.AT23` est défini par
```
DEF=   Nombre(E1(ta1:$))
     + E3(ta2:$).AT32 * E3(ta2:$).AT31
```

[24] En effet, un attribut de base ne dépend d'aucun autre attribut et n'est, par conséquent, destination d'aucun arc.

Le graphe de dépendance complet de ce schéma EA déductif se présente ainsi (les attributs dérivables sont indiqués en italique) :

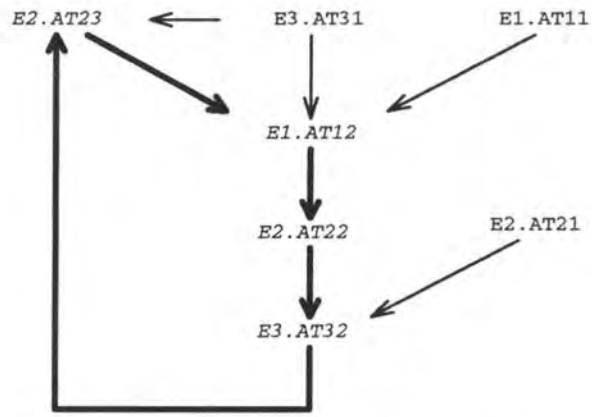


Figure 7-2 : Graphe de dépendance complet

Le graphe de dépendance réduit correspondant se présente comme suit :

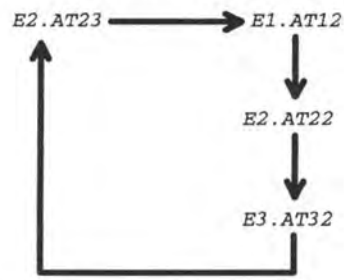


Figure 7-3 : Graphe de dépendance réduit

Dans cet exemple, il est impossible de déterminer les valeurs d'un des attributs dérivables `E2.AT23`, `E1.AT12`, `E2.AT22` ou `E3.AT32` sans avoir préalablement déterminé les valeurs des autres.

Pour détecter la présence de circuits dans un graphe réduit, on utilise un algorithme de décomposition des sommets d'un graphe en niveaux (cfr. [FICHEFET,93]) : on attribue à chaque sommet `s` du graphe réduit[25] un rang établi en ajoutant 1 au maximum des rangs associés aux précédents[26] de `s`. Les sommets sans précédent possèdent le rang 1.

Si `précédent(s)` représente l'ensemble des sommets précédents du sommet `s`, la décomposition en niveaux (attribution d'un rang à chaque sommet) se résume comme suit :

```
Si précédent(s) = ∅ alors rang(s) ← 1
Sinon rang(s) ← 1 + max {rang(t) | t ∈ précédent(s)}
```

Si la décomposition en niveaux réussit (on parvient à attribuer un rang à chaque sommet), le graphe ne possède pas de circuit. Si par contre elle échoue, le graphe possède un circuit.

---

[25] Chaque sommet du graphe correspond à un attribut dérivable.

[26] Un sommet s_1 est précédent d'un sommet s_2 s'il existe dans le graphe un arc reliant s_1 à s_2.

8. *Exploitation du modèle EA déductif*

Nous avons choisi d'exploiter le modèle EA déductif en système définitionnel (cfr. *5.5*) : dans l'exécution d'un schéma, on se sert des formules de définition pour calculer les valeurs des attributs dérivables à partir des valeurs des autres composants du schéma. On se place dans une approche directionnelle où les formules sont vues comme des définitions et non comme des relations ou des équations.

Nous avons vu que, pratiquement, l'exploitation d'un schéma EA déductif en système définitionnel pouvait prendre plusieurs formes (cfr. *5.2*) :

- génération d'une feuille de calcul dans un tableur;
- génération de vues SQL calculant les valeurs des attributs dérivables;
- calcul des valeurs des attributs dérivables et stockage de ces valeurs dans des tables SQL;
- etc.

Dans ce chapitre, nous approfondissons la troisième proposition : calcul des valeurs des attributs dérivables et stockage de celles-ci dans des tables relationnelles. Pour ce qui est du calcul proprement dit des attributs dérivables, plusieurs solutions sont envisageables :

- la première solution serait d'effectuer tous les calculs au sein d'un programme rédigé en un langage classique (C, PASCAL, COBOL, ...) interrogeant le contenu des tables. Pour implémenter une telle solution, il faudrait générer automatiquement, à partir des formules de définition, un programme PASCAL (ou C, COBOL, ...), le compiler et l'exécuter sur la base de données SQL;
- une autre solution serait d'effectuer tous les calculs à l'aide des fonctions de calcul (addition, soustraction, multiplication, division, somme, moyenne, min, max, etc.) offertes par le langage SQL. Pour l'implémentation de cette solution, il faudrait générer automatiquement un script SQL (fichier constitué d'une suite de requêtes SQL) contenant toutes les requêtes nécessaires au calcul et au stockage des valeurs de tous les attributs dérivables et soumettre ce script sur la base de données.

La seconde solution est moins puissante que la première parce que les calculs sont limités aux seules fonctions offertes par SQL alors que la première solution ne pose aucune contrainte sur les possibilités de calcul. Il n'est, par exemple, pas possible d'exprimer en SQL une formule de définition constituée d'une expression logique parce que de telles expressions ne sont pas admises dans la clause `SELECT` d'une requête SQL.

La seconde solution est cependant plus facile à mettre en oeuvre que la première. En effet, il est plus aisé de générer automatiquement un ensemble de requêtes SQL que de générer tout un programme PASCAL. Dans le cadre de ce mémoire, nous optons pour cette seconde solution : calcul et stockage des valeurs des attributs dérivables par un ensemble de requêtes SQL.

Dans ce chapitre, nous détaillons le processus à suivre pour rendre le schéma EA déductif exécutable dans les conditions que nous venons d'établir. La compréhension du processus requiert une bonne connaissance des bases de données relationnelles et, plus

particulièrement, du langage SQL[27]. Nous commençons par donner un aperçu général du processus de transformation.

8.1 Aperçu général du processus de transformation

Les requêtes SQL que nous allons générer pour le calcul des valeurs des attributs dérivables ne s'expriment pas sur le schéma EA déductif mais directement sur la base de données relationnelle correspondante, en terme de tables et de colonnes. Avant de transformer les formules de définition des attributs dérivables en requêtes SQL, il est donc nécessaire de savoir comment le schéma EA déductif est transformé en une base de données relationnelle.

Cette transformation s'opère en deux étapes :

1. le schéma EA déductif (qui se situe au niveau conceptuel) est d'abord transformé en un **schéma conforme au modèle relationnel**. Un schéma EA est conforme au modèle relationnel lorsqu'il ne contient que des structures directement et explicitement exprimables dans le modèle relationnel.

   Exemple : les types d'associations du modèle EA déductif ne sont pas exprimables directement dans le modèle relationnel et doivent être transformés en attributs de référence.

   Le schéma ainsi obtenu est un **schéma logique**.

2. le schéma logique est ensuite transformé en structures compréhensibles par un **Système de Gestion de Base de Données Relationnel** (SGBDR).

   Exemple : On représente chaque type d'entités par une table à laquelle on donne le nom de ce type d'entités.

   Le schéma ainsi obtenu est un **schéma physique**.

Une bonne compréhension des mécanismes de transformation est indispensable pour pouvoir construire, plus tard, des requêtes SQL correctes à partir des formules de définition. Cette succession de transformations peut être schématisée comme suit :

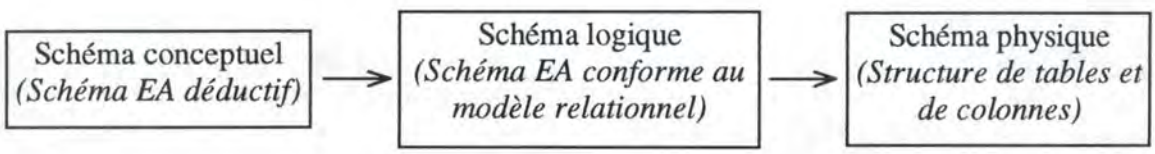


Figure 8-1 : Transformation de schéma

[27] Les ouvrages [HAINAUT,94a] ou [VANDERLANS,93] constituent une bonne introduction aux bases de données relationnelles et au langage SQL.

Dans l'approche choisie, l'objectif final est de stocker les valeurs calculées des attributs dérivables dans des tables SQL indépendantes. Au niveau du schéma physique, on est donc amené à distinguer deux types de tables :

- les **tables de base** qui ne contiennent que des attributs de base;
- les **tables complémentaires** qui contiennent les attributs dérivables : à chaque table de base `T` correspondant à un type d'entités `TE` dans le schéma EA déductif, on associe une table complémentaire `T'` comprenant les valeurs des attributs dérivables de `TE`. Il n'est nécessaire de créer `T'` que s'il y a au moins un attribut dérivable dans `TE`.

Les tables de base ne sont donc pas affectées par les attributs dérivables : toutes les informations dérivées (attributs dérivables) sont placées dans les tables complémentaires. L'exécution d'un schéma EA déductif permet au concepteur d'interroger la base de données grâce au concept d'attribut dérivable sans que la structure des tables de base (tables de production) soit affectée ; on se contente de consulter les tables de base sans modifier ni leur contenu, ni leur structure.

Imaginons, par exemple, le type d'entités `CLIENT` comprenant à la fois des attributs de base et des attributs dérivables. Ce type d'entités donne naissance à deux tables relationnelles [28]:

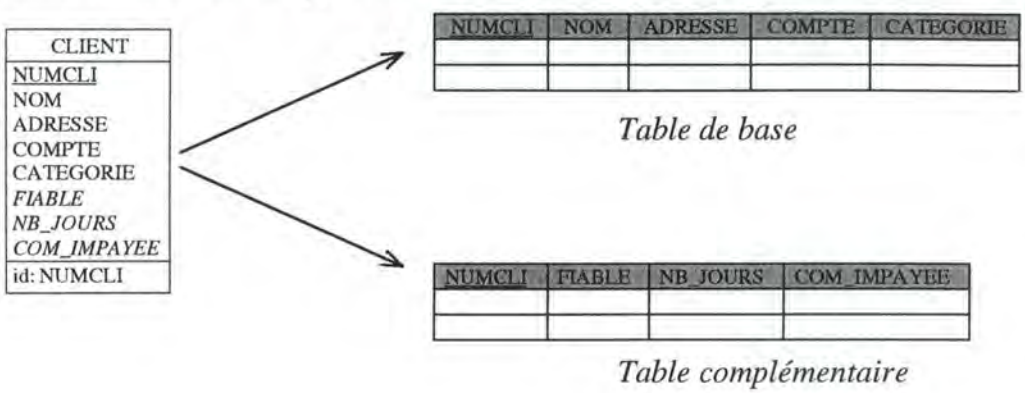


Figure 8-2 : Table de base et table complémentaire

En plus des attributs dérivables, la table complémentaire doit contenir l'identifiant de `CLIENT` : l'identifiant est nécessaire pour retrouver l'entité (le tuple) de la table de base à laquelle se rapportent les informations dérivées contenues dans la table complémentaire.

La transformation *schéma conceptuel* → *schéma logique* → *schéma physique* a pour seul objectif la création du schéma physique correspondant aux tables de base. Lors de cette succession de transformations de schéma, nous ignorons la présence des attributs dérivables (on fait comme s'ils n'existaient pas) : les attributs dérivables n'apparaissent ni dans le schéma logique, ni dans le schéma physique. Ceux-ci ne sont, en effet, pas concernés par la transformation puisqu'ils font l'objet d'un traitement particulier : on crée, pour ces attributs, des tables complémentaires qu'on garnit à l'aide de requêtes SQL.

[28] Les attributs dérivables sont indiqués en italique

L'implémentation physique complète de la base de données relationnelle passe par la rédaction de trois scripts SQL :

1. création des tables de base : ce premier script (script 1) contient toutes les requêtes SQL nécessaires à la création des tables de base. Ces requêtes SQL sont générées à partir du schéma physique associé au schéma EA déductif.
2. création des tables complémentaires : ce second script (script 2) contient toutes les requêtes SQL nécessaires à la création des tables complémentaires.
3. calcul des valeurs des attributs dérivables : ce dernier script (script 3), le plus important, contient toutes les requêtes SQL nécessaires au calcul des valeurs des attributs dérivables et au stockage de ces valeurs dans les tables complémentaires. Ces requêtes SQL sont construites sur base des formules de définition des attributs dérivables.

La soumission du script 1 sur la base de données permet la création des tables de base; celle du script 2, la création des tables complémentaires. Une fois les tables de base garnies de valeurs, on peut soumettre le script 3 qui calcule effectivement les valeurs des attributs dérivables et qui les stocke dans les tables complémentaires. Le processus de transformation complet est synthétisé dans la figure 8-3.

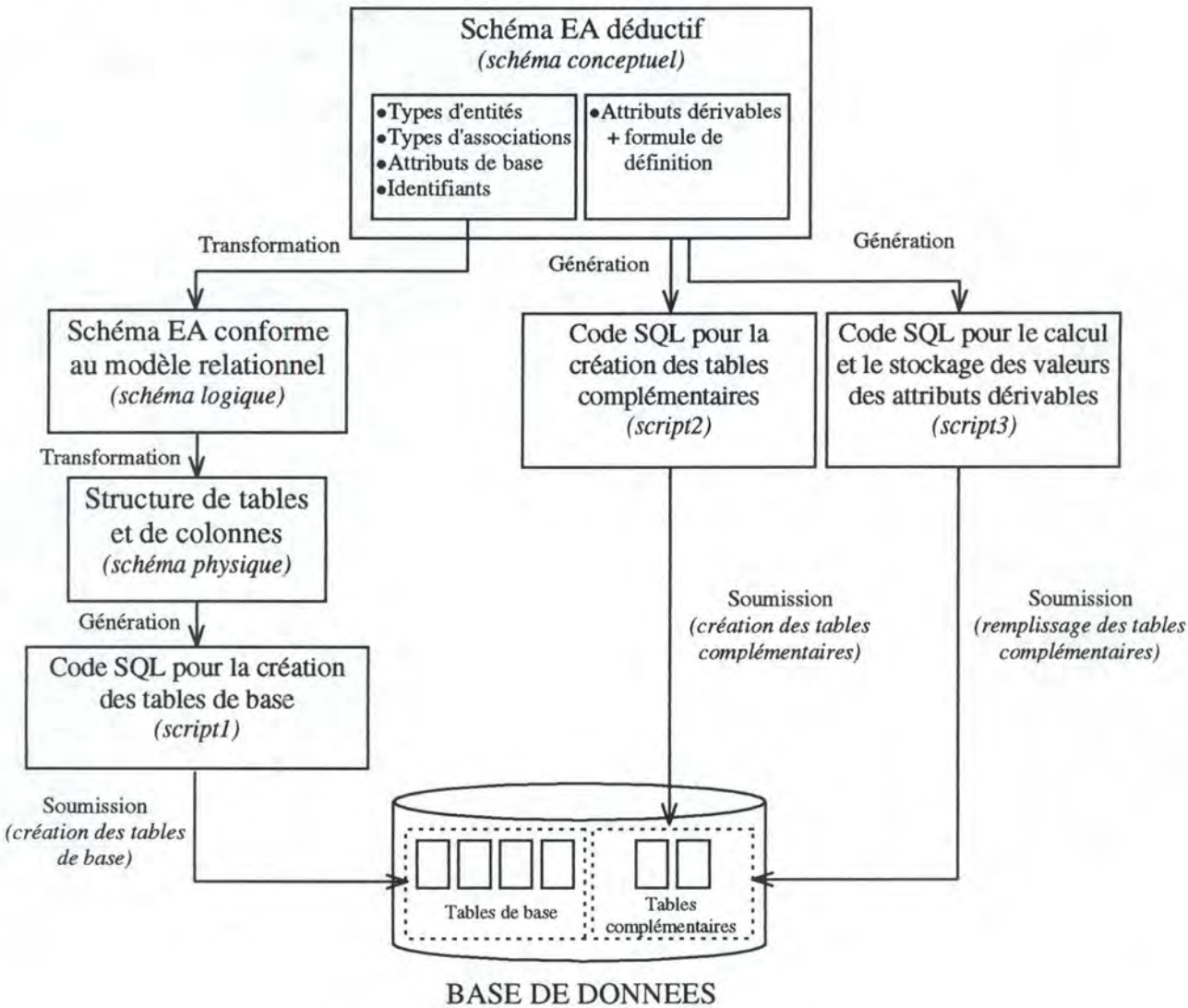


Figure 8-3 : Processus de transformation complet

En cas de modification du contenu des tables de base, les informations des tables complémentaires ne sont plus à jour : pour qu'elles correspondent à nouveau au contenu courant des tables de base, il suffit de rafraîchir les tables complémentaires en soumettant à nouveau le script 3. Ce dernier peut être soumis un nombre illimité de fois (en réalité, à chaque fois qu'on désire mettre à jour les informations dérivées). Les scripts 1 et 2, par contre, ne devraient, normalement, être soumis qu'une seule fois : les tables sont, en effet, créées une fois pour toutes.

En créant une table complémentaire pour chaque table de base, notre solution se rapproche du concept de **Data Warehouse** en vogue dans le domaine de l'aide à la décision : on consulte les données de production stockées dans les tables de base, on les traite en effectuant des calculs dessus et on stocke les résultats (appelés aussi **mesures**) obtenus dans des tables relationnelles complémentaires indépendantes des tables de production (tables de base). Les tables de production ne sont en rien modifiées que ce soit au niveau du contenu ou au niveau de la structure. Dans cette optique, les attributs dérivables peuvent être vus comme un moyen d'interroger, à un niveau conceptuel, les données de production afin d'en dériver de nouvelles informations utiles pour la prise de décision. Pour plus d'informations au sujet des Data Warehouses, nous renvoyons le lecteur à [MORIARTY,95], [WELDON,95] ou encore [DEJESUS,95].

Une approche similaire est suivie par [DECHOW,88] : il propose, sur base d'un modèle EA déductif, de construire une base de données d'aide à la décision (Data Warehouse) garnie en effectuant des extractions et des calculs sur les informations existantes présentes dans la base de données opérationnelle (tables de production).

Nous allons maintenant détailler le processus de transformation :
- nous commencerons par expliquer comment construire les tables de base (cfr. *8.2*);
- nous décrirons ensuite comment construire les tables complémentaires (cfr. *8.3*);
- nous détaillerons enfin le processus de génération des requêtes SQL pour le calcul et le stockage des valeurs des attributs dérivables (cfr. *8.4*).

8.2 Construction des tables de base (script 1)

Comme nous venons de le voir, la construction des tables de base passe par la transformation du schéma déductif en un schéma logique puis en un schéma physique. A partir du schéma physique, il est facile de générer les requêtes SQL permettant la construction physique des tables de base (génération du script 1). Rappelons que les attributs dérivables ne figurent pas dans les tables de base : ils n'entrent pas en ligne de compte dans les transformations successives. Les transformations que nous présentons ici sont extraites de [HAINAUT,94a] et [HAINAUT,94b].

8.2.1 Transformation schéma conceptuel → schéma logique

L'objectif est ici de transformer un schéma EA déductif en un schéma EA conforme au modèle relationnel. Un schéma EA est conforme au modèle relationnel s'il respecte les conditions suivantes :

- le schéma contient des types d'entités, des attributs obligatoires ou facultatifs, des identifiants et des attributs de référence;
- il n'y a aucun type d'associations;

- tout type d'entités possède au moins un attribut;
- tout attribut est monovalué;
- tout attribut est atomique c'est-à-dire non décomposable.

Le modèle EA de base que nous avons défini (cfr. *3.1*) répond aux exigences du modèle relationnel à une exception près : le modèle EA de base accepte les types d'associations alors que ceux-ci sont proscrits dans le modèle relationnel. Il suffit donc de modifier un schéma EA déductif en transformant ses types d'associations pour obtenir un schéma EA logique conforme au modèle relationnel.

Dans la définition du modèle EA standard, nous avons distingué trois classes fonctionnelles pour les types d'associations (cfr. *2.3.1*) : *un-à-un*, *un-à-plusieurs* et *plusieurs-à-plusieurs*. La transformation d'un type d'associations dépend de la classe fonctionnelle à laquelle il appartient.

8.2.1.1 Types d'associations un-à-plusieurs

Soit `ta` un type d'associations un-à-plusieurs entre `A` et `B` (plusieurs `B` pour un `A` et un seul `A` pour chaque `B`). L'objectif de la transformation est d'éliminer le type d'associations `ta` entre `A` et `B` en le remplaçant par une structure admise dans le modèle relationnel. Cette situation pourrait se présenter graphiquement comme suit :

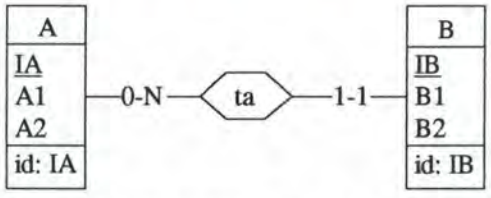


Figure 8-4 : Type d'associations un-à-plusieurs

Pour transformer `ta`, on procède comme suit :

- si `IA` est un attribut identifiant de `A`, on représente `ta` par un attribut `IA'` de même type que `IA` qu'on ajoute à `B` : on ajoute dans `B` l'attribut correspondant à l'identifiant de `A`. Une valeur de `IA'` pour une entité de `B` doit être la référence d'une entité de `A`. Ce nouvel attribut `IA'` dans `B` est appelé **attribut de référence** : il contient pour chaque entité `b` de `B` la référence de l'entité `A` à laquelle `b` est associée [29]. L'attribut de référence `IA'` d'une entité `b` de `B` permet de retrouver l'entité `a` de `A` à laquelle `b` est associée;
- si `ta` est obligatoire pour `B` (la cardinalité minimale de `B` dans `ta` est 1), alors `IA'` doit être déclaré obligatoire. Si, en revanche, `ta` est facultatif pour `B`, `IA'` doit être déclaré facultatif.

Le schéma logique résultant de cette transformation est celui de la figure 8-5 (schéma conforme au modèle relationnel).

---

[29] Toute valeur qu'on trouve pour l'attribut `IA'` d'une entité de `B` doit faire référence à une entité de `A` : la valeur de `IA'` d'une entité quelconque de `B` doit se retrouver comme valeur d'identifiant d'une entité de `A`. Cette règle fait référence au principe d'**intégrité référentielle** : toute valeur prise par un attribut de référence doit être une valeur d'identifiant dans le type d'entités référencé (dans ce cas-ci `A`).

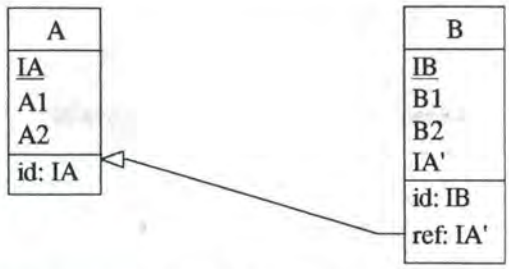


Figure 8-5 : Transformation d'un type d'associations un-à-plusieurs

Le fait que `IA'` de `B` soit un attribut de référence vers `A` est représenté par la flèche partant de `IA'` vers l'identifiant `IA` de `A`.

Si l'identifiant de `A` est constitué de plusieurs attributs IA_1, IA_2, ... , IA_N, on définit un nouvel attribut de référence dans `B` pour chaque attribut de l'identifiant de `A`. Voyons maintenant ce qui se passe quand l'identifiant de `A` est constitué en partie d'autres types d'entités (identifiant hybride).

Pour faciliter la compréhension de ce mécanisme, prenons un exemple concret :

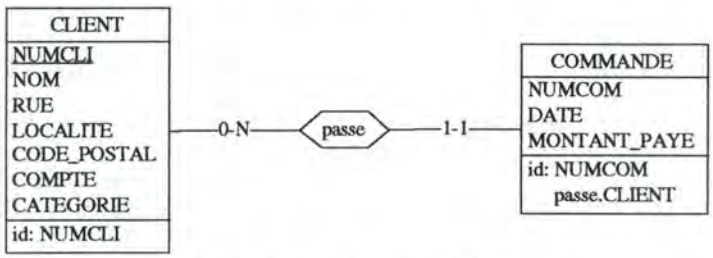


Figure 8-6 : Identifiant hybride

La transformation du type d'associations `passe` entraîne l'ajout de l'attribut de référence `NUMCLI` dans `COMMANDE`. Il est nécessaire d'adapter aussi l'identifiant de `COMMANDE` à cette transformation : on obtient le nouvel identifiant de `COMMANDE` en remplaçant le composant `passe.CLIENT` de l'identifiant par l'attribut de référence qui vient d'être ajouté. Dans notre cas, l'identifiant de `COMMANDE` est, après transformation, constitué des attributs `NUMCOM` et `NUMCLI` comme illustré dans la figure 8-7 (schéma conforme au modèle relationnel).

Figure 8-7 : Transformation d'un identifiant hybride

En reconsidérant le cas général illustré par les figures 8-4 et 8-5, il est important que le nom de l'attribut de référence ajouté dans `B` soit identique au nom de l'attribut de `A` auquel il se

rapporte [30]. S'il existe déjà, dans B, un attribut dont le nom est identique au nom de l'identifiant de A, il est nécessaire de modifier celui-ci avant l'ajout de l'attribut de référence (ou des attributs de référence). En cas de conflit de noms, c'est toujours l'attribut de référence qui l'emporte : ce n'est jamais son nom qui est modifié[31].

Exemple : si dans le schéma de la figure 8-6, il existait déjà dans COMMANDE un attribut nommé NUMCLI, il serait nécessaire de transformer ce nom d'attribut. En effet, c'est l'attribut de référence qui doit être nommé NUMCLI et pas un autre. On transformera, par exemple, l'attribut NUMCLI de COMMANDE par NUMCLIENT avant l'ajout de l'attribut de référence.

8.2.1.2 Types d'associations un-à-un

Soit ta un type d'associations un-à-un entre A et B (un B pour chaque A et un A pour chaque B). Cette situation pourrait se présenter graphiquement comme suit :

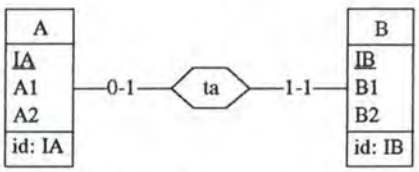


Figure 8-8 : Type d'associations un-à-un

Pour transformer ta, on a le choix entre deux possibilités :

- soit on ajoute à A un attribut similaire à l'identifiant de B qui est en fait un attribut de référence vers B;
- soit on ajoute à B un attribut similaire à l'identifiant de A qui est en fait un attribut de référence vers A.

Si l'identifiant copié est constitué de plusieurs attributs, on crée autant d'attributs de référence qu'il y a d'attributs dans l'identifiant. Si l'identifiant est un identifiant hybride, on procède de la même manière que pour les types d'associations un-à-plusieurs.

Tel que nous venons de le présenter, le mécanisme de transformation est tout à fait aléatoire : on peut choisir le type d'entités dans lequel on met l'attribut de référence. Il est cependant absolument nécessaire de savoir avec précision comment la transformation aura lieu : il faut que celle-ci soit déterministe. On développe, pour ce faire, des règles moins ambiguës :

- si ta est obligatoire pour A et facultatif pour B, on ajoute l'attribut de référence dans A et on le déclare obligatoire;
- si ta est obligatoire pour B et facultatif pour A, on ajoute l'attribut de référence dans B et on le déclare obligatoire;
- si ta est obligatoire pour A et B, alors on ajoute l'attribut de référence dans le type d'entités dont le nom est alphabétiquement inférieur à l'autre et on le déclare obligatoire. Dans notre cas, on ajouterait l'attribut de référence à A (puisque, alphabétiquement parlant, A<B) ;

---

[30] Si l'identifiant de A est constitué de plusieurs attributs IA_1, IA_2, ..., IA_N, il faut que chaque attribut de référence ajouté dans B ait le même nom que l'attribut de A auquel il se rapporte.

[31] Le cas des types d'associations récursifs n'est pas envisagé ici.

- si `ta` est facultatif pour `A` et `B`, alors on ajoute l'attribut de référence dans le type d'entités dont le nom est alphabétiquement inférieur à l'autre et on le déclare facultatif. Dans notre cas, on ajouterait l'attribut de référence à `A`.

Le schéma de la figure 8-8 est, par conséquent, transformé comme suit (schéma conforme au modèle relationnel) :

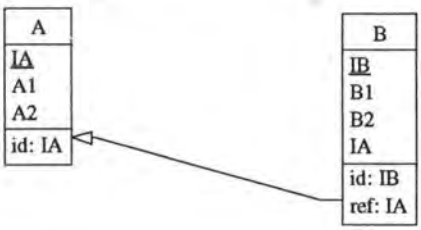


Figure 8-9 : Transformation d'un type d'associations un-à-un

De nouveau, il est important que le nom d'un attribut de référence soit identique au nom de l'attribut identifiant qu'il représente. En cas de conflit de noms, c'est toujours l'attribut de référence qui l'emporte : ce n'est jamais son nom qui est modifié.

8.2.1.3 Types d'associations plusieurs-à-plusieurs

Soit `ta` un type d'associations plusieurs-à-plusieurs entre `A` et `B` (plusieurs `B` pour chaque `A` et plusieurs `A` pour chaque `B`). Cette situation pourrait se présenter graphiquement comme suit :

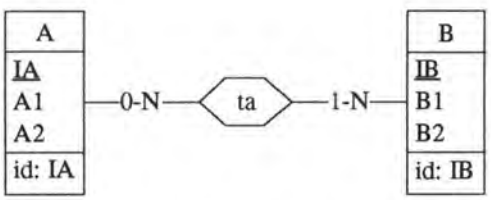


Figure 8-10 : Type d'associations plusieurs-à-plusieurs

On procède d'abord à la transformation de `ta` en un type d'entités de même nom : cette transformation a déjà été introduite en *3.2.3*. On se retrouve alors avec deux types d'associations un-à-plusieurs comme illustré dans la figure suivante :

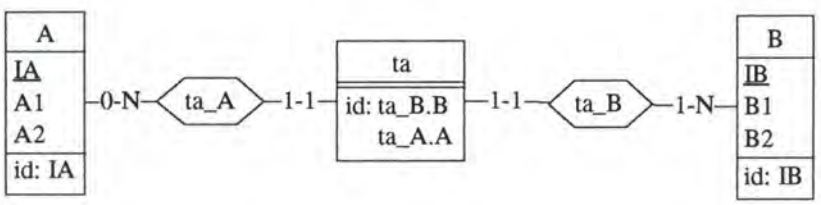


Figure 8-11 : Transformation d'un type d'associations plusieurs-à-plusieurs en un type d'entités

Les deux types d'associations un-à-plusieurs sont ensuite transformés comme indiqué précédemment (cfr. *8.2.1.1*). On obtient finalement le schéma de la figure 8-12, conforme au modèle relationnel.

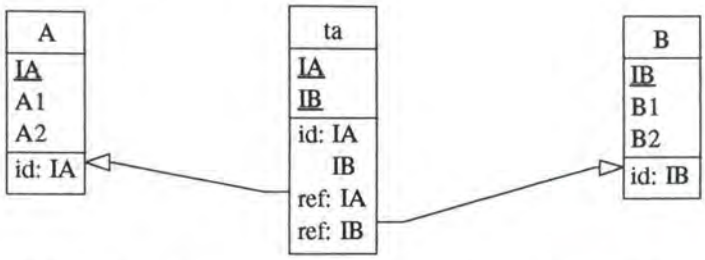


Figure 8-12 : Transformation de deux types d'associations un-à-plusieurs

Le nouveau type d'entités ainsi créé doit posséder le même nom que le type d'associations qu'il représente (dans notre cas, `ta`). De même, les attributs de référence doivent posséder les mêmes noms que les attributs identifiants auxquels ils se rapportent.

8.2.2 Transformation schéma logique → schéma physique

Une fois le schéma EA déductif transformé en un schéma EA conforme au modèle relationnel, il reste à traduire ce dernier en concepts compréhensibles et gérables par un SGBDR. Cette section présente les règles de traduction de chaque type de composants d'un schéma EA (types d'entités, attributs, identifiants et attributs de référence) en structure de données relationnelle (tables, colonnes, identifiants et clefs étrangères).

8.2.2.1 Représentation des types d'entités

On représente chaque type d'entités par une table à laquelle on donne le nom de ce type d'entités.

8.2.2.2 Représentation des attributs

Chaque attribut d'un type d'entités est représenté par une colonne de la table qui représente le type d'entités. On définit son type et sa longueur. Le nom de la colonne est le même que le nom de l'attribut qu'elle représente. La colonne est déclarée obligatoire/facultative (clause `NOT NULL`) selon que l'attribut qu'elle représente est obligatoire/facultatif.

8.2.2.3 Représentation des identifiants

Comme les identifiants des tables sont constitués, dans le schéma logique, uniquement d'attributs (qui sont éventuellement des attributs de référence), la définition des identifiants d'une table est immédiate (clause `PRIMARY KEY (`*colonne*$_1$`,...,`*colonne*$_n$`)`).

8.2.2.4 Représentation des attributs de référence

Les attributs de référence sont représentés comme de simples attributs (cfr. *8.2.2.2*) avec cependant une clause supplémentaire : on les déclare "clefs étrangères" en précisant quelle est la table à laquelle ils font référence (clause `FOREIGN KEY (`*colonne*$_1$`,...,`*colonne*$_n$`) REFERENCES` *table*).

Maintenant qu'on dispose du schéma physique correspondant aux tables de base, il reste à générer les requêtes SQL pour la création de ces dernières (génération du script 1). Le passage du schéma physique aux requêtes SQL étant immédiat, nous ne nous attarderons pas davantage sur la génération du script 1.

8.3 Construction des tables complémentaires (script 2)

Les tables complémentaires sont destinées à stocker les valeurs des attributs dérivables. Nous définissons ici les règles qui permettent de les construire, l'objectif final étant la génération du script 2.

Pour chaque type d'entités `TE` possédant au moins un attribut dérivable, on crée une table complémentaire `COMP_TE` contenant les attributs dérivables de `TE`. Le nom de la table complémentaire est constitué du nom de `TE` préfixé de `"COMP_"`.

Exemple : la table complémentaire de `CLIENT` est nommée `COMP_CLIENT`.

L'identifiant de `COMP_TE` est exactement le même que l'identifiant de la table de base associée à `TE` : pour chaque colonne participant à l'identifiant de la table de base, on crée une colonne similaire (même type, même longueur, même nom) dans `COMP_TE`.

Chaque attribut dérivable de `TE` est représenté par une colonne de `COMP_TE`. On définit son type et sa longueur. Le nom de la colonne est le même que le nom de l'attribut dérivable qu'elle représente. La colonne est déclarée obligatoire/facultative (clause `NOT NULL`) selon que l'attribut dérivable qu'elle représente est obligatoire/facultatif.

A partir de ces règles, il est facile de produire les requêtes SQL pour la création physique des tables complémentaires (génération du script 2) : nous ne nous attarderons pas sur la génération de celles-ci.

8.4 Calcul des valeurs des attributs dérivables (script 3)

Nous expliquons ici comment construire l'ensemble des requêtes SQL qui permettent de calculer les valeurs des attributs dérivables et de stocker ces valeurs dans les tables complémentaires (génération du script 3). Il est nécessaire de générer des requêtes à différents niveaux :

1. requête au niveau des opérandes : la formule de définition d'un attribut dérivable `TE.a` est constituée d'un ensemble d'opérandes (expressions simples) reliés entre eux par des opérateurs. Certains de ces opérandes sont définis par une expression relativement complexe. Afin de simplifier le processus de génération des requêtes, nous isolons le calcul d'un opérande trop complexe `op` dans une requête SQL indépendante : nous créons une **vue SQL** chargée de calculer la valeur de l'opérande `op` pour chaque entité de `TE`.

2. requête au niveau des attributs dérivables : pour chaque attribut dérivable `a` d'un type d'entités `TE`, nous créons une requête chargée du calcul de la valeur de `a` pour chaque entité de `TE`[32]. Ces valeurs sont placées dans une **table temporaire** : on crée une table temporaire pour chaque attribut dérivable `TE.a` du schéma dans laquelle on stocke les valeurs prises par cet attribut pour chaque entité de `TE`. Le calcul des valeurs de `TE.a` utilise les vues définies au niveau 1 pour établir les valeurs de certains opérandes figurant dans la formule de définition de `TE.a`.

[32] Cette requête est construite sur base de la formule de définition de `TE.a`.

3. requête au niveau des tables complémentaires : pour chaque table complémentaire COMP_TE relative à un type d'entités TE, il faut générer une requête SQL qui insère dans COMP_TE les valeurs de tous les attributs dérivables de TE. Ces valeurs sont contenues dans les tables temporaires créées au niveau 2 : on regroupe maintenant ces valeurs dans COMP_TE. On dispose ainsi, pour chaque entité e de TE, de toutes les valeurs dérivées associées à e et cela dans une seule table.

Comme on peut le constater, les calculs des valeurs des attributs dérivables d'un type d'entités TE passent par un ensemble de calculs intermédiaires : au niveau 1, on calcule les valeurs de certains opérandes complexes intervenant dans le calcul des valeurs des attributs dérivables de TE; au niveau 2, on utilise ces résultats pour établir les valeurs des attributs dérivables de TE; au niveau 3, on regroupe finalement les valeurs de tous les attributs dérivables de TE dans la table complémentaire qui lui est associée (COMP_TE).

En supposant que TE_i soit un type d'entités du schéma, que a_j soit un attribut dérivable de TE_i et que op_k soit un opérande complexe figurant dans la formule de définition de $TE_i.a_j$, la décomposition peut se représenter par la figure 8-13.

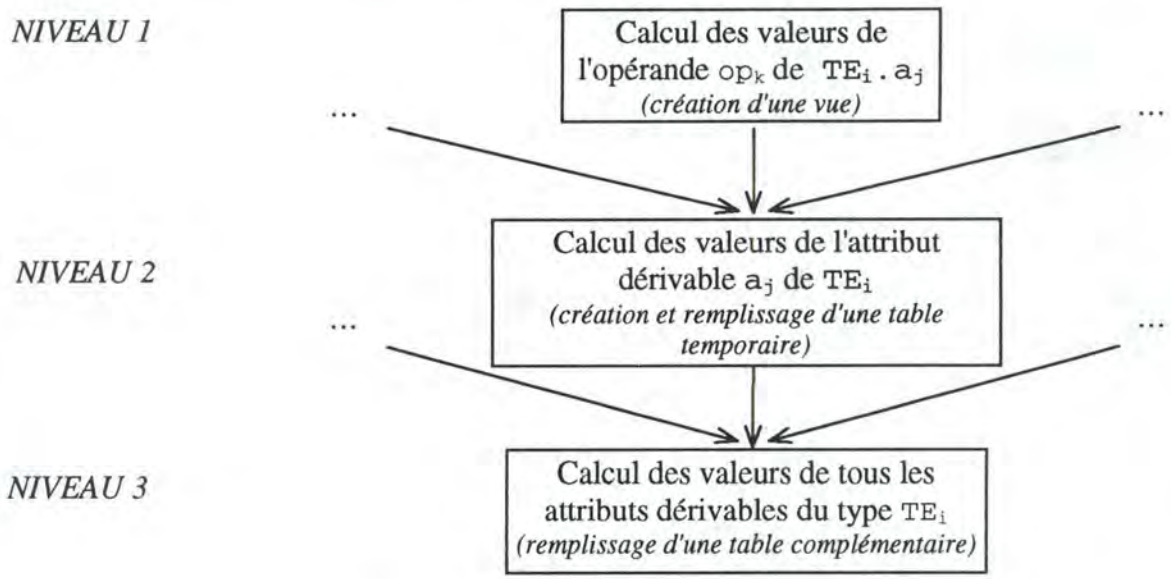


Figure 8-13 : Différents niveaux des requêtes

Une fois les calculs intermédiaires effectués à un certain niveau et stockés dans des tables temporaires ou définis sous la forme de vues, il faut récupérer les résultats au niveau suivant : les requêtes de niveau 2 utilisent les résultats du niveau 1 et les requêtes de niveau 3 utilisent ceux du niveau 2.

Chaque table ou vue intermédiaire créée pour le calcul des valeurs des attributs dérivables d'un type d'entités TE possède comme identifiant le même identifiant que TE ; les valeurs intermédiaires (niveau 1 et 2) doivent, en effet, toujours être calculées pour chaque entité de TE. C'est l'identifiant de TE qui est utilisé pour faire le lien entre les résultats obtenus à un certain niveau et le niveau supérieur (mécanisme de jointure).

Une table temporaire créée au niveau 2 pour le stockage des valeurs d'un attribut dérivable TE.a est nommée "TE_a". Le nom d'une vue créée au niveau 1 pour le calcul d'un

opérande de `TE.a` est obtenu en faisant suivre le nom "`TE_a`" par "_ *i*" où *i* est un numéro de séquence.

Exemple : si le calcul de l'attribut dérivable `FIABLE` de `CLIENT` nécessite la création de deux vues (niveau 1), celles-ci sont nommées `CLIENT_FIABLE_1` et `CLIENT_FIABLE_2`. La table temporaire (niveau 2) associée à l'attribut est nommée `CLIENT_FIABLE`.

Avant d'entamer une description complète de la génération des requêtes à chaque niveau, illustrons l'idée générale du processus par un exemple formel. Un exemple concret plus conséquent est présenté en annexe IV. Considérons le schéma EA déductif suivant (les attributs dérivables sont indiqués en italique) :

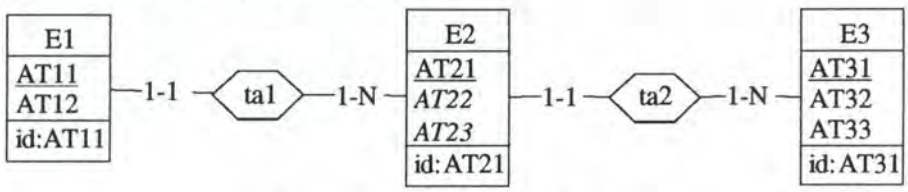


Figure 8-14 : Schéma EA déductif

Les formules de définition sont les suivantes :

- L'attribut `E2.AT22` est défini par `DEF=`
```
E3(ta2:$).AT32 *
Min(I=E1(ta1:$);I.AT12)
```

- L'attribut `E2.AT23` est défini par `DEF=`
```
4 * ($.AT21 +
Somme(I=E3(:AT32 < $.AT21);I.AT32))
```

Le schéma logique correspondant au schéma EA déductif de la figure 8-14 est le suivant (ce schéma ne reprend pas les attributs dérivables) :

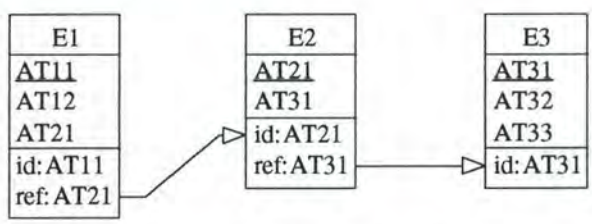


Figure 8-15 : Schéma logique

A partir de ce schéma, le passage au schéma physique (structure de tables et de colonnes) est immédiat : chaque type d'entités donne naissance à une table et chaque attribut, à une colonne. C'est sur le schéma physique que s'expriment les requêtes SQL de calcul.

La table complémentaire associée au type d'entités `E2` est créée par la requête :

```
CREATE TABLE COMP_E2(AT21 NUMERIC(5,0) NOT NULL ,
                     AT22 NUMERIC(5,0) NOT NULL ,
                     AT23 NUMERIC(5,0) NOT NULL ,
                     PRIMARY KEY( AT21 ) );
```

Analysons d'abord la formule de définition de `E2.AT22` : celle-ci est constituée de deux opérandes complexes. Nous décidons de créer une vue calculant les valeurs de chacun d'eux.

La vue relative à l'opérande `E3(ta2:$).AT32` (requête de niveau 1) est définie par :

```
CREATE VIEW E2_AT22_1(AT21,OPERANDE)
 AS
SELECT E2_1.AT21 , E3_1.AT32
FROM   E2 E2_1 , E3 E3_1
WHERE  E2_1.AT31 = E3_1.AT31;
```

La vue relative à l'opérande `Min(I=E1(ta1:$);I.AT12)` (niveau 1) est définie par :

```
CREATE VIEW E2_AT22_2(AT21,OPERANDE)
 AS
SELECT E2_1.AT21 , MIN(E1_1.AT12)
FROM   E2 E2_1, E1 E1_1
WHERE  E2_1.AT21 = E1_1.AT21
GROUP BY E2_1.AT21;
```

Les valeurs de `E2.AT22` sont ensuite stockées dans une table temporaire créée par la requête SQL suivante (requête de niveau 2) :

```
CREATE TABLE E2_AT22  ( AT21 NUMERIC(5,0) NOT NULL,
                        AT22 NUMERIC(5,0) NOT NULL
                        PRIMARY KEY (AT21) );
```

On calcule finalement la valeur de `AT22` pour chaque entité de `E2` en utilisant les résultats intermédiaires fournis par les vues précédentes et on stocke immédiatement ces valeurs dans la table temporaire `E2_AT22` qu'on vient de créer (requête de niveau 2) :

```
INSERT INTO E2_AT22
SELECT E2.AT21,(E2_AT22_1.OPERANDE * E2_AT22_2.OPERANDE)
FROM   E2 E2, E2_AT22_1 E2_AT22_1, E2_AT22_2 E2_AT22_2
WHERE  E2.AT21 = E2_AT22_1.AT21
AND    E2.AT21 = E2_AT22_2.AT21;
```

Analysons maintenant la formule de définition de `E2.AT23` : celle-ci est constituée de trois opérandes. La constante `4` et l'opérande `$.AT21` ne sont pas considérés comme des expressions complexes et ne nécessitent donc pas la création d'une vue.

Nous décidons donc de construire une seule vue pour le calcul de l'opérande `Somme(I=E3(:AT32 < $.AT21);I.AT32)` (requête de niveau 1) :

```
CREATE VIEW E2_AT23_1(AT21,OPERANDE)
 AS
SELECT E2_1.AT21,SUM(E3_1.AT32)
FROM   E2 E2_1 , E3 E3_1
WHERE  E3_1.AT32 < E2_1.AT21
GROUP BY  E2_1.AT21;
```

Les valeurs de `E2.AT23` sont stockées dans une table temporaire créée par la requête SQL suivante (requête de niveau 2) :

```
CREATE TABLE E2_AT23  ( AT21 NUMERIC(5,0) NOT NULL,
                        AT23 NUMERIC(5,0) NOT NULL
                       PRIMARY KEY (AT21) );
```

On calcule finalement la valeur de AT23 pour chaque entité de E2 en utilisant les résultats intermédiaires fournis par la vue précédente et on stocke immédiatement ces valeurs dans la table temporaire E2_AT23 qu'on vient de créer (requête de niveau 2) :

```
INSERT INTO E2_AT23
SELECT E2.AT21,(4 * (E2.AT21 + E2_AT23_1.OPERANDE) )
FROM   E2 E2 , E2_AT23_1 E2_AT23_1
WHERE  E2.AT21 = E2_AT23_1.AT21;
```

Les opérandes $.AT21 et 4 figurent directement dans la requête de niveau 2, sans passer par une vue intermédiaire.

Maintenant qu'on dispose des valeurs de tous les attributs dérivables de E2, on peut regrouper ces valeurs dans la table complémentaire COMP_E2 associée à E2. Avant d'insérer les nouvelles valeurs des attributs dérivables, il est nécessaire de détruire le contenu actuel de COMP_E2. Ces opérations de destruction et d'insertion sont réalisées par les deux requêtes SQL suivantes (requête de niveau 1) :

```
DELETE FROM COMP_E2;

INSERT INTO COMP_E2
SELECT E2.AT21 , E2_AT22.AT22 , E2_AT23.AT23
FROM E2
     LEFT OUTER JOIN E2_AT22 on  E2.AT21 = E2_AT22.AT21
     LEFT OUTER JOIN E2_AT23 on  E2.AT21 = E2_AT23.AT21;
```

Une fois les valeurs de tous les attributs dérivables établies et rangées dans les tables complémentaires, les tables temporaires et les vues ne sont plus nécessaires. Elles sont détruites par les requêtes :

```
DROP TABLE E2_AT23;
DROP VIEW  E2_AT23_1;

DROP TABLE E2_AT22;
DROP VIEW  E2_AT22_2;
DROP VIEW  E2_AT22_1;
```

Comme nous avons opté pour la représentation des formules de définition par des requêtes SQL, il n'est pas possible de représenter toutes les expressions figurant dans une formule : il est nécessaire de restreindre la définition du langage pour adapter celui-ci aux possibilités réduites du langage SQL. Les restrictions apposées sur le langage sont exposées en *8.4.1.* Il est nécessaire, ensuite, d'établir un certain nombre de conventions qui permettent d'éclaircir l'explication des différents processus de génération (cfr. *8.4.2*). Une fois les limitations et les conventions précisées, nous pourrons enfin détailler le processus de génération des requêtes SQL à chaque niveau :

- génération de la requête SQL associée à un opérande (cfr. *8.4.3*);
- génération de la requête SQL associée à un attribut dérivable (cfr. *8.4.4*);
- génération de la requête SQL associée à une table complémentaire (cfr. *8.4.5*).

8.4.1 Limitations

8.4.1.1 Expressions booléennes

La clause `SELECT` d'une requête SQL ne peut pas contenir d'expression booléenne. Il n'est dès lors pas possible de représenter, en SQL, des formules de définition constituées d'une telle expression. Les seules expressions admises sont les expressions simples et les expressions arithmétiques.

8.4.1.2 Ensembles de valeurs

Dans la définition du langage, une condition d'appartenance peut porter sur un ensemble de valeurs de la forme $\{expr_1, \ldots, expr_n\}$ où chaque $expr_i$ $(1 \leq i \leq n)$ est une expression simple (constante, appel de fonction ou désignation d'un attribut). Dans le langage SQL, un ensemble de valeurs ne peut être constitué que de constantes : toutes les expressions simples $expr_i$ dans $\{expr_1, \ldots, expr_n\}$ doivent être des constantes.

Cette restriction ne concerne que le cas où l'ensemble de valeurs est constitué de plusieurs expressions simples placées entre accolades : si l'ensemble de valeurs est constitué d'une seule expression simple (pas d'accolades), celle-ci peut être une constante, un appel de fonction ou une désignation d'attribut.

8.4.1.3 Types d'associations

Dans une expression telle que `TEntité(TAssociation:EnsEnt)` où `EnsEnt` désigne un ensemble d'entités appartenant à un type d'entités `TE`, il faut obligatoirement que `TEntité` soit le nom d'un type d'entités associé via `TAssociation` à `TE`. Dans une expression, on ne peut donc plus, comme le permettait la définition du langage, remplacer le nom d'un type d'entités par le nom du rôle joué par celui-ci dans un type d'associations.

Cette restriction n'est pas liée à une limitation du langage SQL : elle a pour seul objectif de ne pas alourdir les processus de génération que nous exposons dans les sections suivantes. Le principal impact de cette restriction est qu'il devient impossible de faire référence à un type d'associations récursif `ta` puisque l'utilisation de celui-ci dans une formule de définition se faisait en désignant le nom des rôles de `ta` et non `ta` lui-même.

8.4.2 Conventions

8.4.2.1 Représentation des requêtes SQL de sélection

Pour simplifier l'explication de la méthode, nous ne travaillons pas directement sur les requêtes SQL : l'exposé serait, en effet, alourdi inutilement par des détails syntaxiques sans importance. Nous utilisons une représentation des requêtes SQL plus facile à manipuler. Cette représentation concerne uniquement les requêtes de sélection (`SELECT ...`).

On peut voir une requête de sélection comme étant constituée de quatre clauses [33] : `SELECT`, `FROM`, `WHERE` et `GROUP BY`. Nous pouvons donc représenter une telle requête sous la forme d'un quadruplet : `(SELECT, FROM, WHERE, GROUPBY)`.

---

[33] Il existe d'autres clauses telle que `HAVING` ou `ORDER BY` mais celles-ci ne nous sont d'aucune utilité ici.

Chaque composante du quadruplet est constituée d'une liste de valeurs. Pour ajouter un élément quelconque `e` à une liste `l`, on utilise l'opérateur `++` : `l++e` désigne la liste `l` à laquelle on a ajouté l'élément `e`.

Pour désigner une composante de ce quadruplet, on utilise une notation indicée.

Exemple: Si `req` est un quadruplet représentant une requête de sélection, req_{FROM} représente la composante `FROM` de celui-ci.

Considérons la requête SQL suivante :

```
SELECT E2_1.AT21, MIN(E1_1.AT11)
FROM E1 E1_1,E2 E2_1
WHERE E1_1.AT21 = E2_1.AT21
AND E1_1.AT12 < 100
GROUP BY E2_1.AT21 ;
```

Le quadruplet correspondant à celle-ci se présente comme suit :

```
(   [ E2_1.AT21 , MIN(E1_1.AT11)],                    Clause SELECT
    [ (E1,E1_1) , (E2,E2_1)      ],                   Clause FROM
    [ E1_1.AT21 = E2_1.AT21 AND E1_1.AT12 < 100 ],    Clause WHERE
    [ E2_1.AT21                  ] )                  Clause GROUP BY
```

La composante `FROM` du quadruplet est constituée d'un ensemble de couples `(T,A)` où `T` est le nom d'une table et `A` est l'alias qu'on lui associe dans la requête.

Dans le processus de génération des requêtes, à chaque fois qu'une table est ajoutée dans la composante `FROM` d'une requête, on s'arrange pour lui attribuer un nom d'alias non encore utilisé. Pratiquement, le nom de l'alias est constitué du nom de la table suivi d'un numéro de séquence : ceci permet de désigner, de manière non ambiguë, les tables qui apparaîtraient plusieurs fois dans la clause `FROM`.

8.4.2.2 Présentation des exemples

Nous illustrons les différentes étapes du processus de génération par des exemples. Dans chacun de ceux-ci, nous donnons la requête SQL correspondant à une expression du langage. Il est possible, qu'au moment de la présentation de l'exemple, tous les principes de génération nécessaires à la compréhension de celui-ci n'aient pas encore été introduits.

Nous adoptons dès lors la convention suivante dans le texte d'une requête SQL :

- tous les éléments en **`gras`** sont censés être compris par le lecteur parce que le mécanisme nécessaire à l'introduction de cet élément dans le texte de la requête a été expliqué avant l'apparition de l'exemple;
- tous les éléments en `maigre` sont susceptibles de ne pas être compris par le lecteur parce que le mécanisme nécessaire à l'introduction de cet élément dans le texte de la requête n'a pas été expliqué avant l'apparition de l'exemple.

Tous les exemples présentés dans la suite, se réfèrent au schéma EA déductif suivant :

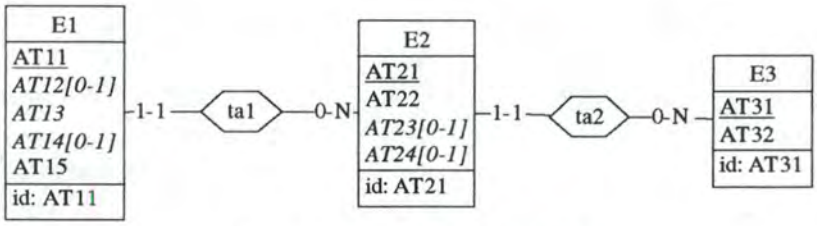


Figure 8-16 : Schéma EA déductif

Voici les formules de définition qui correspondent à ce schéma :

- L'attribut `E1.AT12` est défini par `DEF=`
```
E2(ta1:$).AT22 *
E2(ta1:$).AT23
```

- L'attribut `E1.AT13` est défini par `DEF=`
```
($.AT11 + E2(ta1:$).AT22) * 4
- Min(I=E3;I.AT31)
```

- L'attribut `E1.AT14` est défini par
```
DEF= Moyenne(I=E2(:AT22 = E2(ta1:$).AT22);I.AT21) +
     Moyenne(I=E2(:AT22 > Min(I=E2;I.AT22));I.AT21)
```

- L'attribut `E2.AT23` est défini par `DEF=`
```
$.AT22 + Min(I=E1(ta1:$);I.AT15) -
Somme(I=E2;I.AT22*4)
```

- L'attribut `E2.AT24` est défini par
```
DEF= Max(I=E1(ta1:$ and :AT11>$.AT22);I.AT15) -
     Min(I=E1(ta1:$ or (:AT11>3 and
                 not :AT11 in {1,2,3} ));I.AT15)
```

Les requêtes SQL que nous allons construire s'expriment sur le schéma physique correspondant au schéma EA déductif. En appliquant le mécanisme de transformation *schéma conceptuel → schéma logique* introduit en *8.2.1*, on obtient le schéma logique de la figure 8-17.

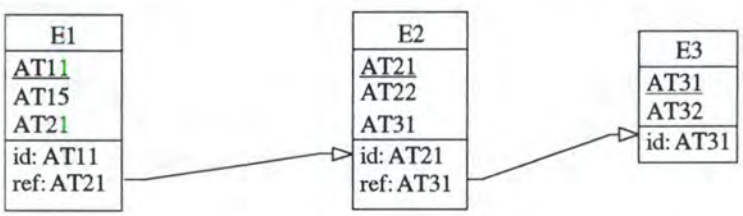


Figure 8-17 : Schéma logique

A partir de celui-ci, le passage au schéma physique est immédiat : chaque type d'entités donne naissance à une table et chaque attribut, à une colonne.

8.4.3 Génération de la requête pour un opérande

La méthode que nous allons développer ici a pour objectif de calculer la valeur d'un opérande `op` d'une formule de définition d'un attribut dérivable `TEntité.a` pour chaque entité de `TEntité`. Pour le calcul de cet opérande, nous créons une vue SQL qui fournit, pour chaque entité `e` de `TEntité`, la valeur de l'opérande `op` correspondant à `e`.

Un opérande est en réalité une expression simple. Dans la définition du langage d'expression des formules de définition, nous avons distingué trois types d'expressions simples (cfr. *4.3.5*) : les expressions de désignation d'un attribut (`Attr`), les appels de fonction (`AppelFct`) et les constantes (`Constante`).

Comme nous l'avons déjà signalé, nous créons une vue uniquement pour un opérande complexe. Une constante n'est pas considérée comme opérande complexe : elle peut être directement spécifiée dans la requête de niveau 2. Il n'est pas nécessaire de créer de vue pour le calcul des constantes.

Dans la génération d'une requête SQL au niveau 1, on distingue dès lors deux types d'opérandes (expressions simples) : les expressions de désignation d'un attribut et les appels de fonction.

Pour exposer la méthode, nous allons parcourir la définition du langage et préciser l'impact de chaque type de constructions sur la requête SQL définissant la vue : pour chaque type de constructions du langage nous définissons un processus chargé d'inclure la sémantique de cette construction dans la requête.

Afin de pouvoir aisément faire référence au processus correspondant à une construction particulière du langage, nous allons nommer chacun de ceux-ci :

- expression simple (`ExprSimple`) → `GEN_VUE_OPERANDE`
- expression de désignation d'un attribut (`Attr`) → `GEN_ATTR`
- appel de fonction (`AppelFct`) → `GEN_FCT`
- expression de désignation d'un ensemble d'entités (`EnsEnt`) → `GEN_ENSENT`
- condition de sélection (`Csel`) → `GEN_CSEL`
- condition d'association (`Cass`) → `GEN_CASS`
- ensemble de valeurs (`EnsVal`) → `GEN_ENSVAL`

8.4.3.1 GEN_VUE_OPERANDE

Pour la création de la vue associée à l'opérande `op` de l'attribut dérivable `TEntité.a`, nous procédons en deux étapes :

1. on crée d'abord la requête de sélection `req` qui va servir à définir la vue;
2. on définit la vue sur base de `req` construite en 1.

Etape 1 : construction de la requête de sélection

La vue que nous devons définir a pour objectif de calculer la valeur de `op` pour chaque entité de `TEntité`.

Il est donc nécessaire :

- d'ajouter le type d'entités `TEntité` [34] dans la composante `FROM` de `req` en lui associant un alias `TEntité'`;
- d'ajouter l'identifiant de `TEntité` dans la composante `SELECT` de `req`.

Si `id(TEntité)` représente l'identifiant de `TEntité`, on initialise donc `req` ainsi :

req_{FROM} ← `(TEntité,TEntité')`
req_{SELECT} ← `id(TEntité)`

Pour obtenir l'identifiant de `TEntité`, on consulte le schéma physique (ou logique) correspondant au schéma EA déductif.

Dans la requête `req`, `TEntité` permet de faire référence à l'entité courante c'est-à-dire l'entité pour laquelle on est en train de calculer la valeur de l'opérande `op`. Lorsqu'on fera allusion à l'entité courante `$` dans une formule de définition, on utilisera `TEntité` (ou, plus précisément, son alias `TEntité'`) dans la requête.

Exemple 1 : Considérons l'opérande `$.AT22` figurant dans la formule de définition de l'attribut dérivable `E2.AT23` et supposons qu'on soit en train de construire une vue `v` pour le calcul de la valeur de cet opérande pour chaque entité de `E2`. La requête de sélection `req` associée à la vue `v` est de la forme :

```
SELECT E2_1.AT21 , E2_1.AT22
FROM E2 E2_1;
```

Poursuivons à présent la description du processus `GEN_VUE_OPERANDE`. Sur base du type de l'expression simple désignée par `op`, on procède maintenant comme suit :

- si l'expression simple est une expression de désignation d'un attribut `Attr`, on appelle `GEN_ATTR` en lui communiquant la requête `req` pour qu'il la modifie de manière à ce que la valeur désignée par `Attr` apparaisse dans le résultat de `req` pour chaque entité de `TEntité`.

- si l'expression simple est un appel de fonction `AppelFct`, on appelle `GEN_FCT` en lui communiquant la requête `req` pour qu'il la modifie de manière à ce que la valeur désignée par `AppelFct` apparaisse dans le résultat de `req` pour chaque entité de `TEntité`. Les fonctions du langage SQL sont des fonctions agrégatives : elles permettent, à partir d'un ensemble de valeurs, de fournir une seule valeur résultat par groupe précisé dans la clause `GROUP BY`. Dans notre cas, il faut une valeur résultat pour chaque entité de `TEntité`. On ajoute donc naturellement l'identifiant de `TEntité` dans la composante `GROUPBY` de `req` :

$req_{GROUPBY}$ ← $req_{GROUPBY}$ `++ id(TEntité)`

---

[34] C'est en réalité la table `TEntité` et non le type d'entités `TEntité` qui doit être ajoutée dans la composante `FROM` de `req`. Pratiquement, cela revient cependant au même puisque le processus de transformation *schéma conceptuel → schéma logique → schéma physique* nous indique qu'à chaque type d'entités du schéma EA déductif correspond en fait une table de même nom dans le schéma physique. Dans l'exposé de la méthode, nous parlerons en terme de type d'entités et non en terme de table en sachant bien que c'est cette dernière qu'on désigne. Ce qui importe, c'est le nom "`TEntité`" à insérer dans la requête et pas le fait que ce soit une table ou un type d'entités.

Etape 2 : Construire la vue associée à l'opérande
On construit une vue sur base de la requête de sélection construite dans l'étape 1. Le nom à attribuer à la vue sera spécifié dans le processus de niveau 2 chargé de la construction de la requête associée à l'attribut dérivable `TEntité.a` : quand le processus de niveau 2 appellera `GEN_VUE_OPERANDE`, il lui communiquera le nom de la vue à créer.

La requête SQL construite en 1 comprend, dans sa composante `SELECT`, l'identifiant de `TEntité` ainsi que l'opérande `op`. Dans la définition de la vue, on nomme les colonnes constituant l'identifiant de `TEntité` par leur nom dans `TEntité` et on nomme `"OPERANDE"` l'opérande calculé par la vue.

Exemple 2 : En reprenant l'exemple 1 ci-dessus, la requête SQL complète permettant la création de la vue associée au calcul de l'opérande `$.AT22` figurant dans la formule de définition de l'attribut dérivable `E2.AT23` est de la forme :

```
CREATE VIEW E2_AT23_1(AT21 , OPERANDE)
 AS
SELECT E2_1.AT21 , E2_1.AT22
FROM E1 E2_1;
```

8.4.3.2 GEN_ATTR

L'objectif du processus `GEN_ATTR` est de modifier la requête `req` reçue du processus appelant pour qu'elle fournisse la (les) valeur(s) désignée(s) par une expression de désignation d'un attribut `Attr`.

Une expression de désignation d'un attribut est toujours de la forme `EnsEnt.Attribut`.

En présence d'une telle construction dans une formule de définition, il faut modifier la requête `req` pour qu'elle fournisse la valeur de `Attribut` pour toutes les entités désignées par `EnsEnt`[35]. Pour ce faire, on appelle d'abord le processus `GEN_ENSENT` en lui communiquant la requête `req` pour qu'il la modifie de manière à ce qu'elle sélectionne les entités représentées par `EnsEnt`.

Ensuite, en supposant que les entités désignées par `EnsEnt` appartiennent au type d'entités `TE` et que `TE'` soit l'alias associé à `TE` dans `req`, il faut modifier la composante `SELECT` de `req` pour que `TE'.Attribut` apparaisse dans le résultat de la requête :

$req_{SELECT} \leftarrow req_{SELECT}$ ++ TE'.Attribut

Exemple 3 : Considérons l'opérande `E2(ta1:$).AT22` figurant dans la formule de définition de l'attribut dérivable `E1.AT12`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E1_1.AT11, E2_1.AT22
FROM E1 E1_1, E2 E2_1                  E2 est ajouté par GEN_ENSENT
WHERE E1_1.AT21 = E2_1.AT21;           Condition de sélection ajoutée par
                                       GEN_ENSENT
```

[35] On suppose que `EnsEnt` désigne plusieurs entités mais le raisonnement est strictement le même dans le cas où `EnsEnt` n'en désigne qu'une seule.

Dans une expression `EnsEnt.Attribut` où les entités de `EnsEnt` sont de type `TE`, si l'attribut `TE.Attribut` est un attribut dérivable, sa valeur ne figure pas dans la table de base `TE` : elle se trouve dans la table temporaire `TE_a` créée au niveau 2. Pour résoudre ce problème de la manière la plus générale et la plus simple possible, on effectue une jointure entre la table `TE` et la table `TE_a` sur base de leur identifiant commun. On dispose ainsi, pour chaque entité e_i de `EnsEnt`, de l'entité correspondante e_i' de `TE_a` et donc de la valeur de l'attribut dérivable `TE.a`. Ceci n'a absolument aucun impact sur le reste du processus de génération : comme l'illustre la figure 8-18, on associe simplement chaque entité de `EnsEnt` (sous-ensemble des entités de `TE`) à son homologue dans la table `TE_a` sans toucher au reste de la requête.

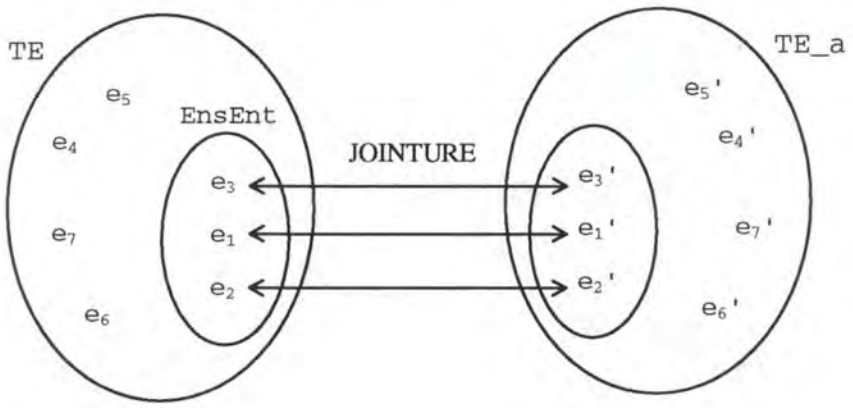


Figure 8-18 : Utilisation d'un attribut dérivable dans une formule de définition

Pratiquement, il suffit d'ajouter la table `TE_a` dans la composante `FROM` de `req` et la condition de jointure entre `TE` et `TE_a` dans la composante `WHERE` de `req`. Dans la composante `SELECT` de `req`, au lieu de préfixer le nom de l'attribut `a` par l'alias correspondant à la table `TE`, on le préfixe par l'alias correspondant à la table `TE_a`.

Exemple 4 : Considérons l'opérande `E2(ta1:$).AT23` figurant dans la formule de définition de l'attribut dérivable `E1.AT12`. L'attribut `E2.AT23` étant dérivable, ses valeurs sont stockées dans la table temporaire `E2_AT23` et non dans la table de base `E2`. La requête de sélection associée à la vue construite pour le calcul de cet opérande s'écrit dès lors comme suit :

```
SELECT E1_1.AT11,E2_AT23_1.AT23
FROM E1 E1_1,E2 E2_1,E2_AT23 E2_AT23_1
WHERE  E2_1.AT21 = E2_AT23_1.AT21     Jointure entre E2 et E2_AT23
AND  E1_1.AT21 = E2_1.AT21;
```

Lors de l'exécution du script 3 (calcul des valeurs des attributs dérivables et remplissage des tables complémentaires), il est nécessaire que la table temporaire `E2_AT23` soit créée et garnie avant l'exécution de la requête ci-dessus. En d'autres mots, il faut ordonner les requêtes dans le script de sorte que les calculs correspondant à un certain attribut dérivable `TEntité.a` aient lieu après les calculs relatifs aux attributs dont dépend `TEntité.a` dans le graphe de dépendance réduit. Nous aurons l'occasion de revenir sur ce point en *8.4.4*.

8.4.3.3 GEN_FCT

L'objectif de GEN_FCT est de modifier la requête req reçue du processus appelant pour qu'elle fournisse la valeur désignée par un appel de fonction Fct. Un appel de fonction peut prendre différentes formes. Les opérations à entreprendre pour la modification de la requête req dépendent du type de l'appel de fonction.

1. Fct est d'une des formes
   Somme(I=EnsEnt;$Expr_I$)
   Min(I=EnsEnt;$Expr_I$)
   Max(I=EnsEnt;$Expr_I$)
   Moyenne(I=EnsEnt;$Expr_I$)
   NombreVal(I=EnsEnt;$Expr_I$)

Dans ce cas, il faut calculer, pour chaque entité I de EnsEnt, la valeur de l'expression $Expr_I$ et ensuite établir la somme, le minimum, le maximum, la moyenne de ces valeurs ou le nombre de valeurs distinctes suivant qu'il s'agit respectivement de la fonction Somme, Min, Max, Moyenne ou NombreVal. Pour ce faire, on appelle d'abord le processus GEN_ENSENT en lui communiquant la requête req pour qu'il la modifie de manière à ce qu'elle sélectionne les entités représentées par EnsEnt.

Ensuite, il faut construire syntaxiquement une expression expr représentant $Expr_I$ pour l'inclure dans la composante SELECT de la requête SQL. On parcourt l'expression $Expr_I$ de gauche à droite et pour chaque composante de $Expr_I$ on procède comme suit :

Si on est sur un opérateur, alors on ajoute simplement l'opérateur à expr.
Sinon, on est sur un opérande o
  Si o est une constante, alors on ajoute o à expr
  Si o est une expression de désignation d'un attribut EnsEnt.Attribut
    alors - il faut modifier req pour qu'elle sélectionne l'entité désignée par EnsEnt : on appelle le processus GEN_ENSENT en lui communiquant la requête req à modifier.
    - en supposant que l'entité de EnsEnt soit du type TE et que TE' soit l'alias attribué à TE, il faut ajouter TE'.Attribut à expr.

Une fois expr construite, on ajoute, à la composante SELECT de req, l'élément SUM(expr), MIN(expr), MAX(expr), AVG(expr) ou COUNT(DISTINCT expr) suivant qu'il s'agit respectivement de la fonction Somme, Min, Max, Moyenne ou NombreVal.

req_{SELECT} ← req_{SELECT} ++ SUM(expr)
req_{SELECT} ← req_{SELECT} ++ MIN(expr)
req_{SELECT} ← req_{SELECT} ++ MAX(expr)
req_{SELECT} ← req_{SELECT} ++ AVG(expr)
req_{SELECT} ← req_{SELECT} ++ COUNT(DISTINCT expr)

Remarque : Si un GROUP BY est nécessaire, celui-ci est ajouté dans la composante GROUPBY de req par le processus GEN_VUE_OPERANDE (cfr. *8.4.3.1*).

Exemple 5 : Considérons l'opérande `Min(I=E1(ta1:$);I.AT15)` figurant dans la formule de définition de l'attribut dérivable `E2.AT23`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E2_1.AT21, MIN(E1_1.AT15)
FROM E2 E2_1, E1 E1_1                    E1 est ajouté par GEN_ENSENT
WHERE E1_1.AT21 = E2_2.AT21              Condition ajoutée par GEN_ENSENT
GROUP BY E2_1.AT21;
```

Exemple 6 : Considérons l'opérande `Somme(I=E2;I.AT22 * 4)` figurant dans la formule de définition de l'attribut dérivable `E2.AT23`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E2_1.AT21, SUM(E2_2.AT22 * 4)
FROM E2 E2_1, E2 E2_2                    E2 est ajouté par GEN_ENSENT
GROUP BY E2_1.AT21;
```

2. `Fct` est de la forme `Nombre(EnsEnt)`
Dans ce cas, il faut calculer le nombre d'entités distinctes de l'ensemble `EnsEnt`.

Considérons que les entités de `EnsEnt` appartiennent au type d'entités `TE`. Si l'identifiant de `TE` dans le schéma logique (ou physique) est constitué d'un seul attribut, on procède comme suit :

- on appelle d'abord le processus `GEN_ENSENT` en lui communiquant la requête `req` pour qu'il la modifie de manière à ce qu'elle sélectionne les entités représentées par `EnsEnt`;
- on modifie ensuite la composante `SELECT` de `req` pour que le nombre d'entités distinctes figurant dans `EnsEnt` apparaisse dans le résultat de `req`. Si l'unique attribut identifiant de `TE` se nomme `Attribut` et si `TE'` est l'alias associé à `TE` dans `req`, on ajoute à la composante `SELECT` de `req` l'élément `COUNT(distinct TE'.Attribut)` :

req_{SELECT} ← req_{SELECT} ++ `COUNT(distinct TE'.Attribut)`

Si par contre, l'identifiant de `TE` dans le schéma logique est constitué de plus d'un attribut, les opérations à exécuter sont plus complexes : on ne peut, en effet, pas spécifier plus d'un attribut dans la clause `COUNT(distinct ...)`.

En supposant qu'on soit en train de définir la formule de définition d'un attribut dérivable `TEntité.a`, la solution consiste à créer une nouvelle vue `v` qui sélectionne, pour chaque entité de `TEntité`, toutes les entités correspondantes de `EnsEnt` et qui ne retient que les couples distincts (`SELECT distinct ...`) associant une entité de `TEntité` à une entité de `EnsEnt`. Au niveau de `req`, on effectue alors une jointure avec `v` sur base de l'identifiant de `TEntité` (on ajoute la condition de jointure dans la composante `FROM` de `req`) et on ajoute un `COUNT(*)` dans la composante `SELECT`. Le `GROUP BY` sur l'identifiant de `TEntité` introduit dans `req` par `GEN_VUE_OPERANDE` permet alors d'obtenir le nombre d'entités distinctes de `EnsEnt` pour chaque entité de `TEntité`.

En résumé, la nouvelle vue `v` sélectionne toutes les entités de `EnsEnt` et ne retient que les couples distincts associant une entité de `TEntité` à une entité de `EnsEnt`. Au niveau de `req`, on se contente uniquement de faire une jointure avec `v` sur base de l'identifiant de `TEntité` et d'ajouter un `COUNT(*)` dans la composante `SELECT` : dans `req`, le

`COUNT(*)` du `SELECT` et le `GROUP BY` sur l'identifiant de `TEntité` permettent d'obtenir le nombre d'entités distinctes de `EnsEnt` pour chaque entité de `TEntité`.

Nous ne détaillerons pas davantage ce processus relativement complexe pour ne pas alourdir l'exposé.

8.4.3.4 GEN_ENSENT

L'objectif du processus `GEN_ENSENT` est de modifier la requête `req` reçue du processus appelant pour qu'elle sélectionne la (les) entité(s) désignée(s) par une expression de désignation d'un ensemble d'entités `EnsEnt`.

Une expression de désignation d'un ensemble d'entités peut prendre plusieurs formes. Les actions à entreprendre dépendent de la forme de `EnsEnt` :

1. `EnsEnt` est de la forme `$`

Dans ce cas, `EnsEnt` désigne l'entité courante. En supposant qu'on soit en train de définir la formule de définition d'un attribut dérivable `TEntité.a`, l'entité courante fait référence à une entité de `TEntité`. Le type d'entités `TEntité` a été ajouté dans la composante `FROM` de `req` par le processus `GEN_VUE_OPERANDE`. Pour faire référence à l'entité courante on utilise l'alias associé à `TEntité` dans `req`. Il n'est donc pas nécessaire de modifier `req` dans ce cas-ci. L'exemple 1 ci-dessus illustre cette première possibilité.

2. `EnsEnt` est de la forme `TE`

Dans ce cas, `EnsEnt` désigne toutes les entités du type `TE`. La requête `req` doit donc sélectionner toutes les entités de ce type : on ajoute dans la composante `FROM` de `req` le type d'entités `TE`. En supposant que `TE'` soit l'alias associé à `TE`,

$$req_{FROM} \leftarrow req_{FROM} \text{ ++ (TE, TE')}$$

L'exemple 6 ci-dessus illustre cette seconde possibilité.

3. `EnsEnt` est de la forme `TE(Csel)`

Dans ce cas, `EnsEnt` désigne toutes les entités du type `TE` qui satisfont à la condition de sélection `Csel`. La requête `req` doit donc sélectionner les entités de `TE` qui satisfont à `Csel` : on commence par ajouter dans la composante `FROM` de `req` le type d'entités `TE`. En supposant que `TE'` soit l'alias associé à `TE`,

$$req_{FROM} \leftarrow req_{FROM} \text{ ++ (TE, TE')}$$

On invoque ensuite le processus `GEN_CSEL` en lui communiquant la requête `req` pour qu'il la modifie de manière à ce que seules les entités de `TE` satisfaisant à `Csel` soient sélectionnées par `req`.

Exemple 7 : Considérons l'opérande `E2(ta1:$).AT22` figurant dans la formule de définition de l'attribut dérivable `E1.AT12`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E1_1.AT11, E2_1.AT22
FROM E1 E1_1, E2 E2_1
WHERE E1_1.AT21 = E2_1.AT21;          Condition ajoutée par GEN_CSEL
```

4. EnsEnt=I

Il s'agit du cas où, dans l'expression $Expr_I$ d'une fonction `Somme(I=EnsEnt';`$Expr_I$`)` (ou `Min`, `Max`, `Moyenne`, `NombreVal`), on utilise l'entité fictive `I` comme expression de désignation d'un ensemble d'entités. En supposant que `EnsEnt'` désigne un ensemble d'entités appartenant au type `TE`, l'entité fictive `I` désigne une entité de ce type. Le type d'entités `TE` a été ajouté dans la composante `FROM` de `req` par le processus `GEN_FCT` qui s'est arrangé pour que `req` sélectionne toutes les entités désignées par `EnsEnt'`. Pour faire référence à l'entité `I`, on utilise l'alias associé à `TE` dans `req`. Il n'est donc plus nécessaire de faire quoi que ce soit à ce niveau-ci. L'exemple 6 ci-dessus illustre cette dernière possibilité.

8.4.3.5 GEN_CSEL

En supposant que `TE` soit le type d'entités sur lequel porte la condition de sélection `Csel`, l'objectif de `GEN_CSEL` est de modifier la requête `req` reçue du processus appelant pour qu'elle sélectionne uniquement les entités de `TE` qui satisfont à `Csel`.

Une condition de sélection `Csel` peut prendre plusieurs formes :

1. `Csel` est une condition d'association unique

Dans ce cas, on appelle le processus `GEN_CASS` en lui communiquant la requête `req` pour qu'il modifie celle-ci de manière à ce qu'elle sélectionne uniquement les entités de `TE` qui satisfont à la condition d'association (`GEN_CASS` modifie la composante `WHERE` de `req`).

2. `Csel` est de la forme $Csel_1$ `AND` $Csel_2$ ou de la forme $Csel_1$ `OR` $Csel_2$

Dans ce cas, on appelle récursivement `GEN_CSEL` pour $Csel_1$ puis pour $Csel_2$. On relie les conditions de sélection correspondant à $Csel_1$ et $Csel_2$ introduites dans la composante `WHERE` de `req` par l'opérateur approprié (`AND` ou `OR`).

3. `Csel` est de la forme `NOT` $Csel_1$

Dans ce dernier cas, on appelle récursivement `GEN_CSEL` pour $Csel_1$ en préfixant la condition de sélection correspondant à $Csel_1$ introduite dans la composante `WHERE` de `req` par l'opérateur de négation `NOT`.

Exemple 8 : Considérons l'opérande `Min( I=E1(ta1:$ or ( :AT11>3 and not :AT11 in {1,2,3} ));I.AT15)` figurant dans la formule de définition de l'attribut dérivable `E2.AT24`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E2_1.AT21, MIN(E1_1.AT15)
FROM E2 E2_1, E1 E1_1
WHERE      E1_1.AT21 = E2_1.AT21      conditions ajoutées par GEN_CASS
OR   (     E1_1.AT11 > 3
       AND NOT (E1_1.AT11 in (1,2,3) )
GROUP BY E2_1.AT21;
```

8.4.3.6 GEN_CASS

En supposant que TE soit le type d'entités sur lequel porte la condition d'association Cass, l'objectif de GEN_CASS est de modifier la requête req reçue du processus appelant pour qu'elle sélectionne uniquement les entités de TE qui satisfont à Cass. Plus particulièrement, une condition d'association a pour effet de modifier la composante WHERE de req puisque c'est dans cette composante que peuvent s'exprimer les conditions d'association.

Une condition d'association Cass peut se présenter sous deux formes différentes :

1. Cass est une condition d'association avec les entités
Dans ce cas, Cass est de la forme :

```
TAssociation:EnsEnt
```

Il faut modifier req de sorte qu'elle sélectionne uniquement les entités de TE associées via TAssociation aux entités désignées par EnsEnt :

- il faut que req sélectionne les entités désignées par EnsEnt. Pour ce faire, on appelle GEN_ENSENT en lui communiquant la requête req à modifier;
- il faut faire en sorte que seules les entités de TE associées via TAssociation à au moins une entité de EnsEnt soient sélectionnées par req. En supposant que les entités de EnsEnt appartiennent au type d'entités TE_2, il faut ajouter à la composante WHERE de req la condition de jointure entre TE et TE_2 :

$req_{WHERE} \leftarrow req_{WHERE}$ ++ CdJointure(TE, TE_2)

La jointure entre les entités de TE et les entités de EnsEnt a pour effet d'associer chaque entité de EnsEnt à l'entité (ou aux entités) correspondante(s) de TE. Toutes les entités de TE n'ayant pas d'entité correspondante dans EnsEnt sont éliminées par la jointure.

Pratiquement, une jointure s'exprime sous la forme d'une (ou de plusieurs) égalité(s) entre colonnes identifiantes et clefs étrangères dans la composante WHERE de la requête (= condition de jointure). On consultera, par exemple, [HAINAUT,94a] pour de plus amples informations sur les jointures. La condition de jointure est construite sur base des identifiants et des clefs étrangères du schéma physique (ou sur base des identifiants et des attributs de référence du schéma logique).

Exemple 9 : Considérons l'opérande E2(ta1:$).AT22 figurant dans la formule de définition de l'attribut dérivable E1.AT12. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E1_1.AT11, E2_1.AT22
FROM E1 E1_1, E2 E2_1
WHERE E1_1.AT21 = E2_1.AT21;  Condition de jointure entre E1 et E2
```

2. Cass est une condition d'association avec des valeurs d'attribut
Dans ce cas, Cass est de la forme :

```
:Attribut Rel EnsVal
```

Il faut modifier `req` de sorte qu'elle sélectionne uniquement les entités de `TE` dont l'attribut `Attribut` respecte la relation `Rel EnsVal` :

- il faut tout d'abord construire la partie de la requête SQL représentant `EnsVal`. On appelle `GEN_ENSVAL` qui nous renverra l'expression SQL `s` correspondant à `EnsVal`. Cette expression peut être une constante, une simple référence à un attribut ou même une sous-requête SQL dans le cas où `EnsVal` est une expression plus complexe (appel de fonction, par exemple);
- en supposant que `TE'` soit l'alias associé à `TE` dans `req`, on ajoute ensuite dans la composante `WHERE` de `req` la condition `TE'.Attribut Rel s` :

req_{WHERE} ← req_{WHERE} ++ (TE'.Attribut Rel s)

Exemple 10 : Considérons l'opérande `Max(I=E1(ta1:$ and :AT11>$.AT22);I.AT15)` figurant dans la formule de définition de l'attribut dérivable `E2.AT24`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E2_1.AT21, MAX(E1_1.AT15)
FROM E2 E2_1, E1 E1_1
WHERE E1_1.AT21 = E2_1.AT21
AND E1_1.AT11 > E2_1.AT22        E2_1.AT22 est construit par GEN_ENSVAL
GROUP BY E2_1.AT21;
```

8.4.3.7 GEN_ENSVAL

L'objectif de ce processus est de construire une expression SQL `s` représentant une expression `EnsVal` et de renvoyer `s`. `EnsVal` peut prendre plusieurs formes :

1. `EnsVal` est une constante `c`

Dans ce cas, l'expression SQL `s` représente simplement `c`. L'exemple 8 ci-dessus illustre cette possibilité.

2. `EnsVal` est un ensemble de constantes $\{c_1, \ldots, c_n\}$

Dans ce cas, l'expression SQL `s` représente l'ensemble de valeurs $(c_1, \ldots, c_n)$. L'exemple 8 ci-dessus illustre cette possibilité.

3. `EnsVal` est une expression de désignation d'un attribut `Attr`

Si `Attr` est de la forme `$.Attribut` et si on considère que l'entité courante est de type `TEntité`, alors il suffit de renvoyer `TEntité'.Attribut` où `TEntité'` est l'alias correspondant à `TEntité`. L'exemple 10 ci-dessus illustre cette première possibilité.

Dans les autres cas, `Attr` est jugé trop complexe et il est nécessaire de construire une sous-requête :

- on crée une nouvelle instance de requête req_2 (nouveau quadruplet vide). Contrairement à ce qui se passe dans `GEN_VUE_OPERANDE` lors de la création de `req`, on n'initialise pas req_2 : on ne met rien dans les composantes de req_2;
- on appelle `GEN_ATTR` en lui communiquant req_2 pour qu'il la modifie de manière à ce qu'elle sélectionne la (les) valeur(s) représentée(s) par `Attr`;
- dans ce cas-ci, `s` est la sous-requête construite par `GEN_ATTR` : on renvoie `s`.

Exemple 11 : Considérons l'opérande `Moyenne(I=E2(:AT22=E2(ta1:$).AT22);I.AT21)` figurant dans la formule de définition de l'attribut dérivable `E1.AT14`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E1_1.AT11, AVG(E2_1.AT21)
FROM E1 E1_1, E2 E2_1
WHERE     E2_1.AT22 = ( SELECT E2_1.AT22
                        FROM E2 E2_1
                        WHERE E2_1.AT21 = E1_1.AT21 )
GROUP BY E1_1.AT11;
```

4. `EnsVal` est un appel de fonction `Fct`

Dans ce cas-ci, on crée systématiquement une sous-requête dont l'objectif est de calculer la valeur représentée par `Fct` :

- on crée une nouvelle instance de requête req_2 (nouveau quadruplet vide);
- on appelle `GEN_FCT` en lui communiquant req_2 pour qu'il la modifie de manière à ce qu'elle calcule la valeur représentée par `Fct`;
- Dans ce cas-ci, `s` est la sous-requête construite par `GEN_FCT` : on renvoie `s`.

Exemple 12 : Considérons l'opérande `Moyenne(I=E2(:AT22 > Min(I=E2;I.AT22));I.AT21)` figurant dans la formule de définition de l'attribut dérivable `E1.AT14`. La requête de sélection associée à la vue construite pour le calcul de cet opérande est de la forme :

```
SELECT E1_1.AT11, AVG(E2_1.AT21)
FROM E1 E1_1, E2 E2_1
WHERE     E2_1.AT22 > ( SELECT MIN(E2_1.AT22)
                        FROM E2 E2_1 )
GROUP BY E1_1.AT11;
```

8.4.4 Génération de la requête pour un attribut dérivable

Pour chaque attribut dérivable `a` d'un type d'entités `TEntité`, il s'agit maintenant de créer une table temporaire et d'y stocker la valeur de `a` pour chaque entité de `TEntité`. Le processus de niveau 2 qui prend en charge ces opérations est nommé `GEN_ATTRDERIV`. Rappelons que la table temporaire créée pour stocker les valeurs de l'attribut `TEntité.a` est nommée `TEntité_a`. `GEN_ATTRDERIV` procède en deux étapes :

1. il génère d'abord une requête SQL pour la création de la table temporaire `TEntité_a`. Cette table temporaire est constituée de l'identifiant de `TEntité` et d'une colonne de même nom que `a` correspondant à l'attribut dérivable [36];
2. il crée ensuite une requête SQL qui insère dans `TEntité_a` la valeur de l'attribut dérivable pour chaque entité de `TEntité`. Il s'agit d'une requête SQL d'insertion (`INSERT INTO TEntité_a SELECT ...`).

---

[36] On obtient l'identifiant de `TEntité` en consultant le schéma logique (ou physique). On crée pour chaque attribut constituant l'identifiant de `TEntité` dans le schéma logique une colonne de même nom que cet attribut dans `TEntité_a`.

La première étape du processus ne posant pas de difficulté particulière, nous détaillons uniquement la seconde. Pour la construction de la requête d'insertion, on crée d'abord une requête de sélection `req`. C'est le résultat de `req` qui est inséré dans `TEntité_a`.

Il faut que `req` fournisse la valeur de `TEntité.a` pour chaque entité de `TEntité`. On initialise donc naturellement `req` comme suit :

req_{FROM} ← `(TEntité,TEntité)`
req_{SELECT} ← `id(TEntité)`

On analyse ensuite la formule de définition de `TEntité.a`. Celle-ci est constituée d'un ensemble d'expressions simples (opérandes) reliées entre elles par des opérateurs arithmétiques. Certains opérandes peuvent être calculés facilement tandis que d'autres sont beaucoup plus complexes. Pour chaque opérande complexe `op`, nous créons une vue au niveau 1 : on appelle le processus `GEN_VUE_OPERANDE` en lui communiquant l'expression simple `op` et le nom de la vue à créer.

Le nom d'une vue créée au niveau 1 pour le calcul d'un opérande de `TEntité.a` est obtenu en faisant suivre le nom "`TEntité_a`" par "_ *i*" où *i* est un numéro de séquence permettant de distinguer les noms des différentes vues créées pour le calcul de `TEntité.a`.

Quand une vue intermédiaire est créée, on intègre ses résultats dans `req` par une jointure entre `TEntité` et la vue sur base de leur identifiant commun (la vue est identifiée par le même identifiant que `TEntité`). Si `NomVue` est le nom attribué à la vue :

req_{FROM} ← req_{FROM} `++ (NomVue,NomVue)`
req_{WHERE} ← req_{WHERE} `++ CdJointure(TEntité,NomVue)`

On réitère ces opérations pour chaque opérande complexe. A chaque fois qu'on traite un opérande `op` (complexe ou non), la composante `SELECT` de `req` est mise à jour en ajoutant l'opérande `op` et en le reliant au reste du `SELECT` par l'opérateur arithmétique approprié.

Remarque : Il est possible que dans la table temporaire `TEntité_a` associée à `TEntité.a` certaines entités de `TEntité` ne soient pas reprises. Considérons, par exemple, l'opérande `E2(ta1:$).AT22` figurant dans la formule de définition de l'attribut dérivable `E1.AT12` et supposons que la cardinalité minimale de `E1` dans `ta1` soit 0. Dans ce cas, il est possible que certaines entités de `E1` ne soient pas associées à une entité de `E2` via `ta1`. En analysant la requête SQL associée à l'opérande `E2(ta1:$).AT22` (cfr. exemple 9 ci-dessus), on constate qu'une entité `e` de `E1` n'ayant pas d'entité correspondante dans `E2` ne figure pas dans le résultat de la vue `v` construite pour le calcul de cet opérande (l'entité `e` est éliminée au moment de la jointure entre `E1` et `E2`). Par conséquent, au moment de la jointure entre `E1` et `v`, la ligne de `E1` correspondant à `e` est éliminée de la jointure et ne figure donc pas dans le résultat de la requête ; `e` n'est pas reprise dans la table temporaire `E1_AT12`.

Déterminons à présent pour chaque type d'opérandes s'il est nécessaire de créer une vue intermédiaire ou si l'opérande peut apparaître directement dans `req` :

- constante : le calcul d'une constante est immédiat. Elle peut donc figurer directement dans `req`.
- expression de désignation d'un attribut : ces expressions peuvent être relativement complexes à calculer. Nous créons pour les opérandes de ce type une vue au niveau 1 sauf lorsque l'attribut concerné est de la forme **$**`.Attribut`. Dans ce cas, on dispose immédiatement de la valeur d'attribut dans la requête de niveau 2 : elle est représentée dans `req` par `TEntité.Attribut`. Il n'est donc pas nécessaire de créer une vue intermédiaire pour son calcul.
- appel de fonction : ces expressions sont généralement complexes à calculer. Nous créons, pour chaque opérande de ce type, une vue intermédiaire.

Une fois la requête de sélection `req` construite, on crée sans peine la requête d'insertion sur base de `req` : on préfixe simplement le texte de la requête représentée par `req` par `"INSERT INTO TEntité_a "`

Exemple : considérons la formule de définition de l'attribut dérivable `E1.AT13`. Celle-ci est constituée de 4 opérandes. Pour les opérandes `E2(ta1:$).AT22` et `Min(I=E3;I.AT31)` nous créons deux vues au niveau 1 nommées respectivement `E1_AT13_1` et `E1_AT13_2`. Les définitions de ces vues sont les suivantes :

```
CREATE VIEW E1_AT13_1(AT11,OPERANDE)
 AS
SELECT E1_1.AT11,E2_1.AT22
FROM E1 E1_1,E2 E2_1
WHERE  E1_1.AT21 = E2_1.AT21;

CREATE VIEW E1_AT13_2(AT11,OPERANDE)
 AS
SELECT E1_1.AT11, MIN(E3_1.AT31)
FROM E1 E1_1,E3 E3_1
GROUP BY E1_1.AT11;
```

Au niveau 2, on commence par créer une table temporaire `E1_AT13` (étape 1) :

```
CREATE TABLE E1_AT13(AT11 NUMERIC(5,0) NOT NULL,   Identifiant de E1
                     AT13 NUMERIC(5,0),            Nom de l'attribut dérivable
                     PRIMARY KEY (AT11));
```

On insère ensuite dans `E1_AT13` les valeurs de `AT13` pour chaque entité de `E1` (étape 2) :

```
INSERT INTO E1_AT13
SELECT    E1.AT11,
          (E1.AT11 + E1_AT13_1.OPERANDE) * 4 - E1_AT13_2.OPERANDE
FROM E1 E1, E1_AT13_1 E1_AT13_1, E1_AT13_2 E1_AT13_2
WHERE  E1.AT11 = E1_AT13_1.AT11          Jointure entre E1 et E1_AT13_1
AND  E1.AT11 = E1_AT13_2.AT11;           Jointure entre E1 et E1_AT13_2
```

Au moment de l'exécution des requêtes associées au calcul d'un attribut dérivable `TEntité.a` (exécution de la requête de niveau 2 qui fait appel aux vues de niveau 1 pour le calcul de certains opérandes), il faut disposer des valeurs des attributs dérivables dont dépend `TEntité.a` dans le graphe de dépendance réduit. Dans le script 3, il est donc

nécessaire d'ordonner les requêtes SQL de niveau 1 et 2 de sorte que les requêtes relatives au calcul d'un attribut dérivable `TEntité.a` soient placées, dans le script, après toutes les requêtes relatives au calcul des attributs dérivables dont dépend `TEntité.a`.

Pour ordonner les requêtes, on se base sur la décomposition en niveaux des sommets du graphe de dépendance réduit (cfr. *7.2.2.1*) : c'est en fonction du rang attribué à chaque attribut dérivable (sommet) lors de la décomposition, qu'on place les requêtes de niveau 1 et 2 dans le script 3 (on place d'abord les requêtes relatives aux attributs dérivables de rang 1, puis de rang 2, etc.).

8.4.5 Génération de la requête pour une table complémentaire

Une fois que toutes les tables temporaires correspondant aux attributs dérivables d'un type d'entités `TEntité` ont été créées et remplies, on peut regrouper les valeurs des attributs dérivables dans la table complémentaire `COMP_TEntité` associée à `TEntité`. Le processus au niveau 3 qui prend en charge ces opérations est nommé `GEN_TABCOMP`. Il procède en deux étapes :

1. il génère d'abord une requête SQL chargée de détruire le contenu de la table `COMP_TEntité` : le contenu actuel de celle-ci est remplacé par les nouvelles valeurs des attributs dérivables (on procède à un rafraîchissement de la table complémentaire);
2. il crée ensuite une requête SQL qui insère dans `COMP_TEntité` les valeurs de chaque attribut dérivable de `TEntité`. Il s'agit d'une requête SQL d'insertion (`INSERT INTO COMP_TEntité SELECT ...`).

La première étape ne pose pas de difficulté particulière. Attardons-nous sur la seconde. Pour la construction de la requête d'insertion, on crée une requête de sélection `req` : c'est le résultat de `req` qui est inséré dans `COMP_TEntité`.

Il faut que `req` fournisse les valeurs de chaque attribut dérivable pour chaque entité de `TEntité`. On initialise donc naturellement `req` comme suit :

req_{FROM} ← (TEntité, TEntité)
req_{SELECT} ← id(TEntité)

Le type d'entités `TEntité` possède un certain nombre d'attributs dérivables $a_1, \ldots, a_n$. Les valeurs de chaque attribut dérivable `TEntité.`a_i ($1 \leq i \leq n$) sont contenues dans la table temporaire `TEntité_`a_i au niveau 2.

Pour chaque attribut dérivable `TEntité.`a_i, on effectue, dans `req`, une jointure entre `TEntité` et la table temporaire `TEntite_`a_i (niveau 2) sur base de leur identifiant commun :

req_{FROM} ← req_{FROM} ++ (TEntité_a_i,TEntité_a_i)
req_{WHERE} ← req_{WHERE} ++ CdJointure(TEntité,TEntité_a_i)

La jointure entre `TEntité` et `TEntité_`a_i possède une particularité : il s'agit d'une **jointure externe** (*Outer Join*). Il faut, en effet, disposer d'une ligne dans `COMP_TEntité` pour chaque entité de `TEntité`. Or, nous avons vu qu'il est possible que, pour une entité `e` de `TEntité`, il n'existe pas de ligne correspondant à `e` dans la table temporaire

TEntité_a_i (cfr. *8.4.4*). Si on se contente d'une jointure classique (*Inner Join*) entre TEntité et TEntité_a_i, la ligne associée à e ne figure pas dans le résultat de la requête req et n'apparaît donc pas dans la table complémentaire COMP_TEntité.

Avec une jointure classique, il suffit qu'il y ait une table temporaire TEntité_a_i ($1 \leq i \leq n$) dans laquelle une entité e de TEntité n'est pas reprise pour que e n'apparaisse pas dans COMP_TEntité même si e possédait des valeurs pour les autres attributs dérivables TEntité.a_j ($1 \leq j \leq n$ et $j \neq i$) ; e est éliminée par la jointure entre TEntité et TEntité_a_i.

Avec une jointure externe, on est certain de retrouver une ligne pour chaque entité e de TEntité dans le résultat de la requête indépendamment du fait que e soit reprise ou non dans chaque table temporaire TEntité_a_i ($1 \leq i \leq n$) : si e ne possède pas de valeur pour un attribut dérivable a_i de TEntité, la valeur de e dans la colonne correspondant à a_i dans COMP_TEntité est NULL.

On ajoute finalement le nom de l'attribut dérivable dans la composante SELECT de req :

$req_{SELECT} \leftarrow req_{SELECT}$ ++ TEntité_a_i.a_i

On réitère ces opérations pour chaque attribut dérivable de TEntité. Une fois la requête de sélection req construite, il est trivial de construire la requête d'insertion complète : on préfixe simplement le texte de la requête représentée par req par "INSERT INTO COMP_TEntité".

Quand toutes les tables complémentaires ont été garnies, on peut détruire les vues et les tables temporaires.

Exemple : Considérons le type d'entités E1. Celui-ci possède trois attributs dérivables AT12, AT13 et AT14. Pour chacun de ceux-ci, on dispose d'une table temporaire créée au niveau 2 qui contient les valeurs des attributs dérivables pour les entités de E1 (ou, plus précisément, pour une partie des entités de E1). Ces tables sont nommées respectivement E1_AT12, E1_AT13 et E1_AT14. Au niveau 3, on commence par effacer le contenu de la table complémentaire COMP_E1 associée à E1 :

```
DELETE FROM COMP_E1;
```

On insère ensuite les valeurs des attributs dérivables de E1 dans COMP_E1 :

```
INSERT INTO COMP_E1
SELECT E1.AT11 , E1_AT12.AT12 , E1_AT13.AT13 , E1_AT14.AT14
FROM E1
     LEFT OUTER JOIN E1_AT12 on E1.AT11 = E1_AT12.AT11
     LEFT OUTER JOIN E1_AT13 on E1.AT11 = E1_AT13.AT11
     LEFT OUTER JOIN E1_AT14 on E1.AT11 = E1_AT14.AT11;
```

Une fois que toutes les tables complémentaires ont été garnies, on peut détruire les vues et les tables temporaires :

```
DROP TABLE E1_AT12;
DROP VIEW E1_AT12_2;
DROP VIEW E1_AT12_1;

DROP TABLE E1_AT13;
DROP VIEW E1_AT13_2;
DROP VIEW E1_AT13_1;

DROP TABLE E1_AT14;
DROP VIEW E1_AT14_2;
DROP VIEW E1_AT14_1;
```

Rappelons, pour conclure ce chapitre, qu'un exemple complet (exemple non formel) se trouve en annexe IV.

9. *Implémentation*

La grande majorité des concepts et processus développés dans les chapitres précédents ont été implémentés. L'application que nous avons créée permet, à partir d'un schéma EA déductif, d'effectuer une étude de cohérence de ce schéma et de le rendre exécutable dans le cadre des options choisies (génération automatique de scripts SQL en système définitionnel).

L'objectif de ce chapitre est d'expliquer l'architecture globale de l'application. Une description plus détaillée des modules et le code source des programmes peuvent être consultés dans le volume II de ce mémoire.

9.1 Environnement de développement

L'application a été implémentée dans l'outil DB-MAIN ([DBMAIN,95]) développé à l'Institut d'Informatique de l'Université de Namur par l'équipe du Professeur J-L. Hainaut. DB-MAIN est un outil *CASE* dédié à l'ingénierie de bases de données et plus particulièrement à la conception des bases de données, au *reverse engineering* et à la maintenance des bases de données.

DB-MAIN possède de nombreuses fonctionnalités. Dans le cadre de ce mémoire, nous n'utiliserons qu'une partie de celles-ci :

- conception et gestion de schémas EA standards;
- mécanismes de transformation *schéma conceptuel* → *schéma logique*;
- mécanismes de transformation *schéma logique* → *schéma physique*.

DB-MAIN offre la possibilité d'associer une description sémantique à chaque composant d'un schéma EA : c'est au niveau de la description sémantique d'un attribut dérivable qu'on spécifie sa formule de définition. DB-MAIN offre donc un environnement logiciel permettant d'exprimer parfaitement tous les concepts du modèle EA déductif.

De plus, DB-MAIN est accompagné d'un langage de programmation comparable aux langages classiques tel que C ou PASCAL : *Voyager 2* ([ENGLEBERT,95]). L'application développée dans le cadre de ce mémoire a été rédigée dans ce langage.

Voyager 2 offre des fonctions qui permettent de manipuler directement le référentiel (*repository*) de DB-MAIN. Il permet donc facilement d'obtenir toutes les informations relatives à un schéma EA déductif : quels sont les types d'entités du schéma? Quels sont les attributs d'un type d'entités? Quels sont les attributs dérivables? Quelle est la formule de définition d'un attribut dérivable? Un programme *Voyager 2* compilé peut être immédiatement exécuté à partir de DB-MAIN.

9.2 Architecture globale

L'application est constituée de 3 programmes :

- `ANALSYN` : réalise une analyse lexicale et une vérification syntaxique des formules de définition d'un schéma;
- `ANALSEM` : réalise une vérification sémantique des formules de définition;
- `GENERAT` : génère les scripts SQL pour le calcul des valeurs des attributs dérivables.

L'architecture globale du système est illustrée par la figure 9-1.

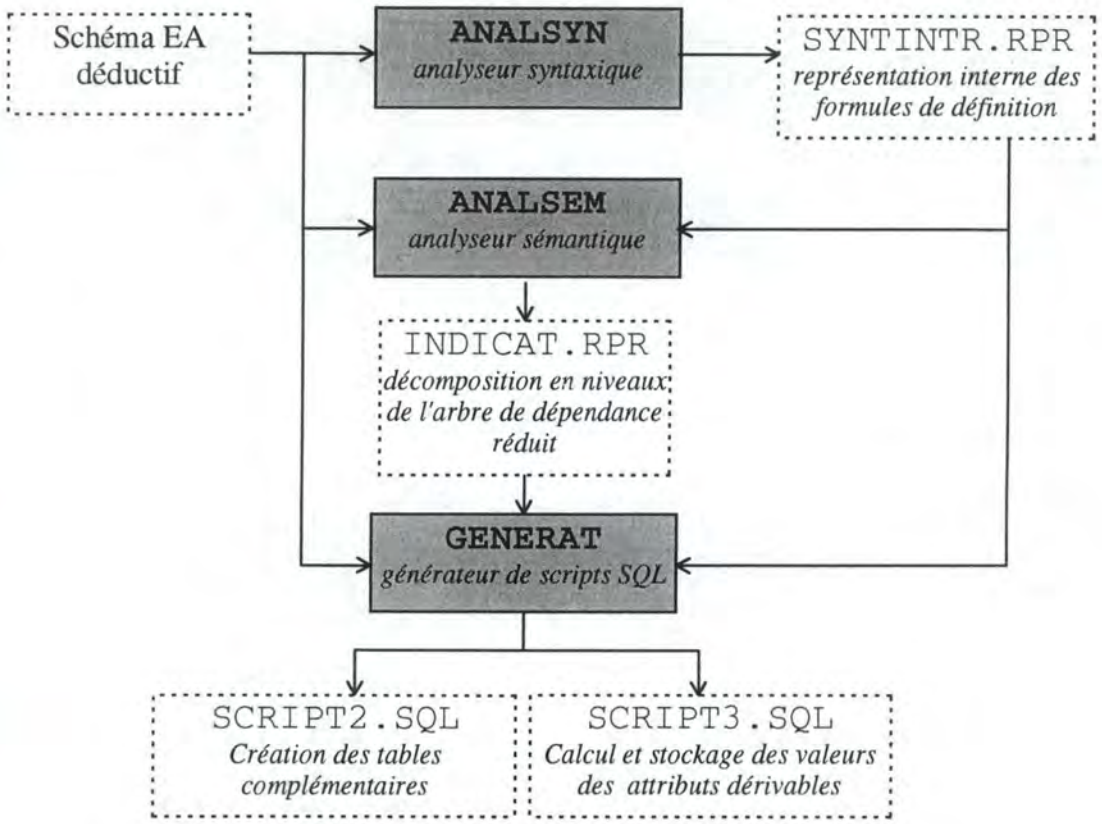


Figure 9-1 : Architecture globale du système

Détaillons à présent chaque programme en décrivant ses principales caractéristiques, les informations qu'il utilise en entrée et celles qu'il produit en sortie.

9.2.1 Analyseur syntaxique : ANALSYN

L'analyseur syntaxique a pour objectif de vérifier que toutes les formules de définition des attributs dérivables du schéma sont **cohérentes au niveau syntaxique** (cfr. *7.1*) c'est-à-dire qu'elles respectent les spécifications du langage d'expression des formules de définition.

ANALSYN consulte, en entrée, le schéma EA déductif pour en extraire les formules de définition des attributs dérivables. Si chaque formule est syntaxiquement correcte, il crée le fichier de sortie SYNTINTR.RPR contenant la **représentation interne** des formules de définition : l'analyseur sémantique (ANALSEM) et le générateur (GENERAT) ne travaillent plus directement sur le texte des formules de définition tel qu'il se trouve dans le schéma déductif mais sur une représentation interne de celles-ci, plus facile à manipuler[37].

Toute incohérence syntaxique détectée par ANALSYN est communiquée au concepteur : celui-ci est alors invité à corriger les éventuelles erreurs et à relancer ANALSYN jusqu'à obtenir un schéma EA déductif cohérent au niveau syntaxique.

L'analyseur syntaxique utilise un analyseur lexical (contenu également dans ANALSYN) pour décomposer une formule de définition en symboles de base.

---

[37] La spécification de la représentation interne des formules de définition se trouve dans le volume II.

9.2.2 Analyseur sémantique : ANALSEM

L'analyseur sémantique a pour objectif de vérifier que toutes les formules de définition du schéma EA déductif sont cohérentes d'un point de vue sémantique (cfr. *7.2*). ANALSEM procède en deux phases :

- dans la première, il vérifie la **cohérence sémantique locale** (cfr. *7.2.1*) : cohérence des types, cohérence des noms des attributs, des types d'associations et des types d'entités, cohérence des expressions de type `Attr` et cohérence des valeurs `NULL` et attributs facultatifs[38];
- dans la seconde, il vérifie la **cohérence sémantique globale** (cfr. *7.2.2*) en s'assurant qu'il n'y a pas de circuit dans le graphe de dépendance réduit du schéma. Pour ce faire, il utilise un algorithme de détection de circuit basé sur la décomposition en niveaux des sommets du graphe.

ANALSEM travaille sur la représentation interne des formules de définition construite par ANALSYN : ANALSEM utilise SYNTINTR.RPR en entrée. Il consulte également le schéma EA déductif pour vérifier, par exemple, la cohérence des noms des types d'associations.

Toute incohérence sémantique détectée par ANALSEM est communiquée au concepteur : celui-ci est alors invité à corriger les éventuelles erreurs et à relancer d'abord ANALSYN (pour construire une nouvelle représentation interne), puis ANALSEM.

ANALSEM fournit, en sortie, le fichier INDICAT.RPR contenant la décomposition en niveaux des sommets du graphe de dépendance réduit : ce fichier contient pour chaque sommet (c'est-à-dire, pour chaque attribut dérivable), le rang attribué à celui-ci (cfr. *7.2.2.1*).

9.2.3 Générateur de scripts : GENERAT

Le générateur a pour objectif de créer les scripts SQL qui rendent exécutable le schéma EA déductif (cfr. *8*) :

- SCRIPT2.SQL : ce script contient toutes les requêtes SQL nécessaires à la création des tables complémentaires (correspond au script 2 dans le chapitre *8*);
- SCRIPT3.SQL : ce script contient toutes les requêtes SQL nécessaires au calcul des valeurs des attributs dérivables et au stockage de ces valeurs dans les tables complémentaires (correspond au script 3 dans le chapitre *8*).

Pour la génération de ces scripts, GENERAT se base sur la représentation interne des formules de définition (fichier SYNTINTR.RPR). Il utilise aussi la décomposition en niveaux des sommets du graphe de dépendance réduit (INDICAT.RPR) pour l'ordonnancement des requêtes SQL au sein du script 3. Il consulte finalement le schéma EA déductif pour en déduire, par exemple, l'identifiant d'un type d'entités.

Les scripts sont générés pour être exécutés avec le SGBDR *InterBase* de *Borland* ([BORLAND,95]).

---

[38] ANALSEM n'effectue pas une analyse de cohérence au niveau des unités.

Le script 1 correspondant à la création des tables de base est généré automatiquement par DB-MAIN lui-même : il a suffi de modifier le programme de génération pour qu'il ignore les attributs dérivables afin que ceux-ci ne figurent pas dans les tables de base.

10. Extensions

Dans ce chapitre, nous proposons quelques pistes pour prolonger le travail commencé dans ce mémoire :

- extension du modèle EA déductif;
- extension de l'étude de cohérence;
- extension du mode d'exploitation;
- autres modes d'exploitation du modèle EA déductif.

10.1 Extension du modèle EA déductif

Au niveau des informations dérivables, le modèle EA déductif permet uniquement la représentation d'attributs dérivables. On pourrait étendre les spécifications du modèle pour permettre l'expression des formules de définition d'autres éléments dérivables d'un schéma : types d'entités dérivables et types d'associations dérivables.

Le modèle EA déductif est défini comme une extension du modèle EA de base qui est lui-même un sous-ensemble des formalismes utilisés dans le modèle EA standard. On pourrait étendre les spécifications du modèle EA déductif pour y intégrer des mécanismes de modélisation plus puissants qu'on trouve dans le modèle EA standard : types d'associations de degré supérieur à 2, types d'associations avec attributs, sous-types, cardinalités `[i-j]` quelconques, attributs multivalués et décomposables, etc.

On pourrait étendre les spécifications du langage d'expression des formules de définition pour enrichir le concept d'entité courante. Ainsi, comme nous l'avons mentionné en *4.5*, on devrait pouvoir désigner, dans une condition de sélection, l'entité courante de tout type d'entités intermédiaire apparaissant dans la formule de définition d'un attribut dérivable `TE.a` et pas uniquement l'entité courante du type d'entités `TE`. Le langage pourrait également être enrichi par un plus grand nombre d'opérateurs et de fonctions.

Exemple : concaténation de chaînes de caractères (opérateur '+' pour les chaînes), fonctions statistiques (écart type, variance), fonctions de manipulation de chaînes (transformer majuscules/minuscules, tester l'appartenance d'une suite de caractères à une chaîne), etc.

10.2 Extension de l'étude de cohérence

Comme nous l'avons déjà introduit en *7.2.1.5*, on pourrait effectuer une analyse de cohérence au niveau des unités des attributs mises en correspondance dans les formules de définition. Pour cela, il faudrait associer à chaque attribut, dérivable ou non, l'unité dans laquelle il s'exprime et établir une table de correspondance entre unités afin de savoir :

- si, au sein des formules de définition, les unités des attributs sont compatibles;
- comment effectuer des conversions d'unités (on transforme, par exemple, des grammes en Kg en les divisant par 1000 dans la formule).

10.3 Extension du mode d'exploitation

Dans le mode d'exploitation que nous avons choisi, on génère, à partir d'un schéma EA déductif, un ensemble de requêtes SQL qui permettent le calcul des valeurs des attributs dérivables. Plus précisément, c'est la soumission du script 3 sur la base de données (cfr. *8*)

qui permet le calcul des valeurs dérivables et le stockage de ces valeurs dans des tables complémentaires. L'exécution du script 3 déclenche d'office le calcul des valeurs de tous les attributs dérivables.

On pourrait imaginer un système de génération dans lequel l'utilisateur peut choisir les attributs dérivables dont il désire rafraîchir les valeurs : on créerait alors un script contenant uniquement les requêtes SQL nécessaires au calcul et à la mise à jour (dans les tables complémentaires) des valeurs des attributs dérivables sélectionnés par l'utilisateur.

10.4 Autre mode d'exploitation

Comme nous l'avons déjà signalé, le modèle EA déductif est indépendant des différentes exploitations qu'on peut en faire. Dans ce mémoire nous avons approfondi un seul mode d'exploitation : génération de requêtes SQL pour le calcul et le stockage des valeurs des attributs dérivables dans des tables relationnelles complémentaires.

Il existe de nombreuses autres exploitations possibles du modèle :

- génération d'une feuille de calcul dans un tableur (EXCEL, LOTUS 123, etc.) : on effectue tous les calculs des attributs dérivables dans la feuille de calcul en reliant celle-ci à une base de données relationnelle pour disposer des valeurs des attributs de base. L'exploitation du modèle EA déductif par un tableur est parfaitement adaptée en système définitionnel. De plus, la plupart des tableurs proposent aujourd'hui des résolveurs d'équations : ceux-ci sont particulièrement utiles pour l'exploitation du modèle déductif en système relationnel;
- génération d'un programme C, PASCAL, COBOL, etc. qui calcule les valeurs des attributs dérivables en consultant une base de données relationnelle et qui stocke les valeurs de ceux-ci dans des fichiers ou dans des tables relationnelles complémentaires. Contrairement à la création de scripts SQL, la construction d'un programme de $3^{ème}$ génération n'impose aucune restriction sur les formules de définition puisque la puissance de calcul n'est pas limitée (exemple : on peut, dans un langage de $3^{ème}$ génération, exprimer sans problème des formules de définition constituées d'une expression booléenne alors que SQL n'admet pas de telles formules);
- génération d'un modèle OMP pour le calcul des valeurs des attributs résultats;
- etc.

Nous expliquons maintenant comment il est possible d'exploiter le modèle EA déductif d'une autre manière que par des scripts SQL : on donne des directives sommaires concernant la génération d'un modèle OMP à partir d'un schéma EA déductif pour le calcul des attributs résultats.

10.4.1 Générateur d'un modèle OMP

OMP est un progiciel d'optimisation qui implémente l'algorithme de programmation linéaire du Simplexe. Il dispose d'un générateur de modèles qui permet une formulation relativement aisée des problèmes de programmation linéaire ainsi qu'une exécution de ceux-ci. Pour plus d'informations au sujet d'OMP nous renvoyons le lecteur à [OMP,87].

Dans cette section, nous donnons quelques propositions pour la génération d'un modèle OMP qui rend exécutable un schéma EA déductif.

OMP permet l'exploitation d'un schéma EA déductif en système définitionnel (cfr. *5.2*), en système relationnel (cfr. *5.3*) ou en optimisation (cfr. *5.4*). Notre objectif est de donner ici une ébauche de spécification pour l'exploitation du modèle EA déductif à l'aide de modèles OMP : nous nous limitons à l'exposé des principes de base.

Nous nous plaçons d'abord dans le cas le plus simple qui est l'exploitation, par un modèle OMP, d'un schéma EA déductif en système définitionnel (cfr. *10.4.1.1*) : les formules sont vues comme des définitions et non comme des relations ou des équations. Nous dirons ensuite deux mots sur l'exploitation, par un modèle OMP, d'un schéma EA déductif en système relationnel (cfr. *10.4.1.2*).

10.4.1.1 Système définitionnel : principes de base

Dans le système définitionnel, les attributs données et les attributs résultats sont fixés à l'avance : à partir d'un schéma EA déductif, on génère un modèle OMP qui permet le calcul des valeurs des attributs dérivables à partir des valeurs des attributs de base.

Un modèle OMP est constitué de plusieurs parties : le scénario, l'objectif, la définition des *sets*, la définition des variables, la définition des contraintes et la définition des données.

Pour chaque type d'entités `TE`, on crée un fichier de données OMP qui comprend la valeur de chaque attribut de base pour chaque entité de `TE` : chaque ligne du fichier représente une entité de `TE` et chaque colonne représente un attribut de base de `TE`. Ce fichier pourra aisément être construit à partir d'une extraction des informations stockées dans une base de données de production.

Dans le modèle OMP, on définit, pour chaque type d'entités, un *set* comprenant toutes les entités de ce type (clause `SET= ...`). Les entités de `TE` sont, par exemple, nommées `te1`, `te2`, `...`, `teN` dans le *set* correspondant. La définition du *set* repose généralement sur les informations présentes dans le fichier de données OMP associé au type d'entités `TE`.

Pour chaque attribut de base de `TE`, on définit dans le modèle OMP une donnée (clause `DATA= ...`) en utilisant le *set* correspondant à `TE`.

Pour chaque attribut dérivable `TE.a` du schéma EA déductif, on définit une variable (clause `X=...`) dans le modèle OMP en utilisant le *set* correspondant à `TE`. La formule de définition correspondant à cette variable est traduite par une contrainte (clause `C=...`).

L'exécution du modèle par OMP calcule la valeur de tous les attributs dérivables $TE.a_i$ pour chaque entité de `TE`.

En système définitionnel, chaque attribut dérivable $TE.a_i$ prend une valeur unique pour chaque entité de `TE`. La fonction objectif peut donc indifféremment être une maximisation ou une minimisation puisque dans les deux cas, les valeurs des attributs dérivables sont les mêmes. Chaque variable correspondant à un attribut dérivable intervient dans la fonction objectif avec un coefficient 0.

Le modèle OMP est linéaire alors que le modèle EA déductif ne l'est pas du tout. Il ne sera dès lors pas possible d'exprimer, dans le modèle OMP, une formule de définition contenant le produit de deux attributs dérivables puisque à chaque attribut dérivable est associé une

variable dans le modèle OMP et que le produit de ces deux variables rendrait le modèle non-linéaire.

Donc, si TA1, TA2 et TA3 sont trois attributs dérivables de TE et si la formule de définition de TE.TA3 est de la forme DEF= $.TA1*$.TA2, alors cette formule de définition ne saura pas être exprimée dans le modèle OMP. Il s'agit donc de restreindre fortement la définition du langage d'expression des formules de définition pour tenir compte de cette limitation. Avant la génération d'un modèle OMP à partir d'un schéma EA déductif, il faut vérifier qu'aucune formule de définition ne contient un produit de deux attributs dérivables.

Détaillons à présent les principes de base en traitant un exemple complet.

Exemple
Considérons le schéma EA déductif suivant :

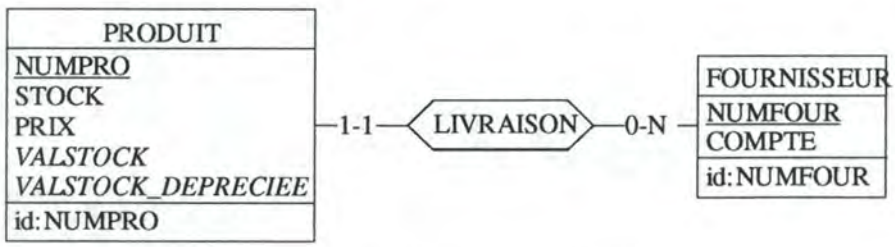


Figure 10-1 : Schéma EA déductif DISTRIBUTION

- l'attribut dérivable VALSTOCK de PRODUIT représente la valeur du stock pour un produit. Il est défini par :
  DEF= $.STOCK * $.PRIX

- l'attribut dérivable VALSTOCK_DEPRECIEE de PRODUIT représente la valeur du stock dépréciée de 10%. Il est défini par :
  DEF= $.VALSTOCK * 9/10

On associe au type d'entités PRODUIT un fichier comprenant toutes les valeurs des attributs de base pour toutes les entités de PRODUIT. Dans le modèle OMP, ce fichier est nommé FI_PRODUIT. Le fichier relatif au type d'entités FOURNISSEUR est nommé FI_FOURNI. Ces deux fichiers se présentent comme suit :

FI_PRODUIT

	NUMPRO	STOCK	PRIX
produit1			
produit2			
...			
produitN			

FI_FOURNI

	NUMFOUR	COMPTE
fournisseur1		
fournisseur2		
...		
fournisseurM		

Figure 10-2 : Format des fichiers FI_PRODUIT et FI_FOURNI

On définit ensuite un *set* reprenant toutes les entités de PRODUIT. Pour la définition de ce *set*, on se base sur la première colonne du fichier FI_PRODUIT :

```
SET= PRODUIT, FICHIER= FI_PRODUIT, LIGNE=1
```

On procède de même pour le type d'entités FOURNISSEUR :

```
SET= FOURNISSEUR, FICHIER= FI_FOURNI, LIGNE=1
```

Pour chaque attribut de base de PRODUIT et de FOURNISSEUR on définit maintenant les données :

```
DATA= PRODUIT_NUMPRO, FICHIER=FI_PRODUIT, L=PRODUIT(&), C=NUMPRO
DATA= PRODUIT_STOCK, FICHIER=FI_PRODUIT, L=PRODUIT(&), C=STOCK
DATA= PRODUIT_PRIX, FICHIER=FI_PRODUIT, L=PRODUIT(&), C=PRIX
DATA= FOURNISSEUR_NUMFOUR, FICHIER=FI_FOURNI, L=FOURNISSEUR(&), C=NUMFOUR
DATA= FOURNISSEUR_COMPTE, FICHIER=FI_FOURNI, L=FOURNISSEUR(&), C=COMPTE
```

Pour chaque attribut dérivable de PRODUIT, on définit une variable continue dans le modèle. Chaque variable intervient dans la fonction objectif avec un coefficient nul :

```
X=PRODUIT(&).VALSTOCK                = C   $0
X=PRODUIT(&).VALSTOCK_DEPRECIEE      = C   $0
```

Pour la formule de définition de chaque attribut dérivable du schéma EA déductif, on définit une contrainte dans le modèle OMP :

- La formule de définition de PRODUIT.VALSTOCK est constituée du produit de deux attributs de base. OMP ne permet pas d'exprimer le produit de deux données dans une contrainte. Il est dès lors nécessaire de définir une donnée intermédiaire (nommée, par exemple, PRIXSTOCK) pour le calcul du produit :

```
DATA=PRIXSTOCK, TP= PRODUIT_STOCK * PRODUIT_PRIX
C= CALCUL.PRODUIT(&).VALSTOCK =
   PRODUIT(&).VALSTOCK = /PRIXSTOCK/
```

- La formule de définition de PRODUIT.VALSTOCK ne pose aucun problème particulier :

```
C= CALCUL.PRODUIT(&).VALSTOCK_DEPRECIEE =
   PRODUIT(&).VALSTOCK_DEPRECIEE = 0.9 * PRODUIT(&).VALSTOCK
```

Il reste maintenant à définir le scénario :

```
SCENARIO = EXECUTION DU SCHEMA DISTRIBUTION
```

et la fonction objectif :

```
MIN
```

Le modèle OMP complet se présente comme suit :

```
*****************************************
SCENARIO = EXECUTION DU SCHEMA DISTRIBUTION
*****************************************
MIN
*
* Definition des sets
*
SET= PRODUIT,     FICHIER= FI_PRODUIT,    LIGNE=1
SET= FOURNISSEUR, FICHIER= FI_FOURNI,     LIGNE=1
*
* Definition des variables
*
X=PRODUIT(&).VALSTOCK                = C   $0
X=PRODUIT(&).VALSTOCK_DEPRECIEE      = C   $0
*
* Contraintes
```

```
*
C= CALCUL.PRODUIT(&).VALSTOCK =
   PRODUIT(&).VALSTOCK = /PRIXSTOCK/

C= CALCUL.PRODUIT(&).VALSTOCK_DEPRECIEE =
   PRODUIT(&).VALSTOCK_DEPRECIEE = 0.9 * PRODUIT(&).VALSTOCK
*
* Definition des donnees relatives aux types d'entités
*
DATA= PRODUIT_NUMPRO, FICHIER=FI_PRODUIT, L=PRODUIT($), C=NUMPRO
DATA= PRODUIT_STOCK, FICHIER=FI_PRODUIT, L=PRODUIT($), C=STOCK
DATA= PRODUIT_PRIX, FICHIER=FI_PRODUIT, L=PRODUIT($), C=PRIX
DATA= FOURNISSEUR_NUMFOUR, FICHIER=FI_FOURNI, L=FOURNISSEUR($), C=NUMFOUR
DATA= FOURNISSEUR_COMPTE, FICHIER=FI_FOURNI, L=FOURNISSEUR($), C=COMPTE
*
* Definition des donnees intermediaires
*
DATA=PRIXSTOCK, TP= PRODUIT_STOCK * PRODUIT_PRIX
*
```

10.4.1.2 Système relationnel : principes de base

En système relationnel, un attribut dérivable ou un attribut de base peut jouer tantôt le rôle d'attribut donnée, tantôt celui d'attribut résultat. Lors de la génération du modèle OMP, il est nécessaire que l'utilisateur précise quels attributs jouent le rôle de données et quels attributs jouent le rôle de résultats. Pour chaque attribut donnée, on définit une donnée (`DATA=...`) dans le modèle OMP et pour chaque attribut résultat, on définit une variable (`X=...`).

Pour la suite de la génération du modèle OMP, on procède, à peu de choses près, de la même manière qu'en système définitionnel : on définit un *set* pour chaque type d'entités et une contrainte pour chaque formule de définition[39].

[39] Lors de l'écriture de la contrainte, il faut tenir compte, d'un point de vue syntaxique, du rôle joué par chaque attribut apparaissant dans la formule de définition : certains sont traduits par des variables, d'autres par des données. Dans un modèle OMP, les données sont entourées du symbole '/'.

Conclusion

Le modèle EA est, à l'heure actuelle, un des modèles conceptuels le plus utilisé dans la modélisation de la structure des informations. Dans ce travail, nous avons étendu le modèle EA afin de le doter de la possibilité d'exprimer le caractère dérivable de certaines informations ; le modèle EA déductif permet de modéliser le fait que certaines informations sont déduites à partir d'autres.

Le modèle EA déductif est muni d'un langage permettant l'expression des formules de définition des informations dérivables. Notre objectif était d'obtenir une implémentation opérationnelle du modèle EA déductif. Afin d'y parvenir dans les délais impartis, nous nous sommes limités, d'une part, à une version simplifiée du modèle EA standard (modèle EA de base) et, d'autre part, à un langage simple permettant uniquement la modélisation des attributs dérivables. Ce langage possède certaines lacunes : nous avons proposé des moyens pour contourner ses limites et nous avons suggéré plusieurs voies pour enrichir sa spécification. Malgré son pouvoir d'expression restreint, l'utilisation du modèle EA déductif convient parfaitement à un utilisateur aux exigences modérées.

Une fois le langage d'expression des formules de définition spécifié, nous avons analysé différentes pistes pour exploiter le modèle EA déductif : à partir d'un schéma EA déductif, on implémente un système capable de calculer effectivement les informations dérivées (on rend le schéma exécutable). Nous avons envisagé l'exploitation du modèle en mode définitionnel, en mode relationnel et en optimisation. Notre choix s'est finalement porté sur le mode définitionnel.

Nous avons ensuite proposé un ensemble de règles de cohérence que tout schéma EA déductif doit respecter pour avoir un comportement correct. Le respect de ces règles est nécessaire mais n'est pas suffisant : il est possible qu'un schéma EA déductif respectant les règles produise quand même des résultats erronés. La liste des règles n'est pas exhaustive ; on peut la compléter notamment par l'ajout d'un contrôle sur les unités.

Nous avons alors spécifié et développé un système capable de rendre un schéma EA déductif exécutable : à partir d'un tel schéma, notre système génère automatiquement un des requêtes SQL qui permettent le calcul effectif des valeurs des attributs dérivables dans une base de données relationnelle en système définitionnel. Les valeurs des attributs dérivables sont mises à la disposition des utilisateurs dans des tables SQL indépendantes des tables de production (tables de base).

La création de tables indépendantes peut être comparée au concept de *Data Warehouse* : on consulte les données de production stockées dans les tables de base, on les traite et on stocke les résultats obtenus (appelés aussi **mesures**) dans des tables relationnelles complémentaires, indépendantes des tables de production. Les tables de production ne sont en rien modifiées que ce soit au niveau du contenu ou au niveau de la structure. Les requêtes SQL générées permettent un rafraîchissement des informations dérivables quand nécessaire.

Le système de génération automatique de scripts SQL ainsi que les analyseurs syntaxiques et sémantiques des formules de définition ont été implémentés dans l'atelier DB-MAIN d'ingénierie de bases de données. L'intégration du modèle EA déductif dans DB-MAIN permet la spécification et la génération de SIAD aux possibilités relativement modestes : pour prendre leurs décisions, les gestionnaires se basent, entre autres, sur les informations

opérationnelles présentes dans la base de données de production. Le modèle EA déductif implémenté dans DB-MAIN leur permet d'exprimer les informations dont ils ont besoin à ce niveau pour la prise de décision. Le concept d'attribut dérivable leur offre la possibilité d'interroger, à un niveau conceptuel, le contenu d'une base de données opérationnelle afin d'en inférer les mesures pertinentes dans leur activité décisionnelle.

Le schéma EA déductif se situant au niveau conceptuel, l'utilisateur du modèle (le gestionnaire) dispose d'une vue des données facile à comprendre et à manipuler ; il n'est pas encombré par des détails techniques d'implémentation.

Le système implémenté dans DB-MAIN n'est qu'un prototype. Celui-ci souffre de plusieurs lacunes :

- les performances du système lors du calcul effectif des informations dérivées (soumission des scripts) sont relativement mauvaises;
- le système ne permet pas de rafraîchir uniquement une partie des attributs dérivables : lors de la soumission des scripts, toutes les informations dérivables sont systématiquement recalculées alors qu'un calcul partiel est suffisant dans la plupart des cas (le gestionnaire se contente souvent d'une partie seulement des informations dérivées).
- le système n'est pas capable de rendre exécutable n'importe quel schéma EA déductif : à cause, notamment, des limitations du langage SQL, toutes les formules de définition des attributs dérivables ne sont pas implémentables (expressions booléennes, ensembles de valeurs, etc.).

Le prototype devrait être amélioré à ces points de vue. Pour ce qui est des performances, une analyse plus approfondie des requêtes SQL à générer doit être entreprise. Nous pensons que d'autres modes d'exploitation du modèle EA déductif fourniraient de meilleurs résultats : génération de programmes C, COBOL ou PASCAL, génération de feuilles de calcul dans un tableur, etc. De plus, un calcul partiel des seules informations dérivables nécessaires aurait un impact très positif au niveau des performances. Malgré ses défauts, le système a cependant le mérite d'être opérationnel et de fournir des résultats corrects.

Pour terminer, nous voudrions insister sur le fait que le modèle EA déductif spécifié dans ce mémoire est totalement indépendant des différentes exploitations qu'on peut en faire. Nous avons proposé de rendre un schéma EA déductif exécutable par la génération de scripts SQL et nous avons donné une ébauche de spécification pour la génération d'un modèle OMP. D'autres modes d'exploitations du modèle EA déductif sont envisageables ; nous en avons proposé certains et il en existe certainement davantage. Nous sommes donc loin d'avoir abordé tous les aspects du problème : il existe de nombreuses voies pour poursuivre le travail entrepris.

Bibliographie

[BATINI,92], Batini C., Ceri S., Navathe S., *"Conceptual Database Design, An Entity-Relationship Approach"*, Benjamin/Cummings, 1992.

[BLANING,87], Blaning R., *"A relational Theory of Model Management"*, in Decision Support Systems : Theory and Application, Holsapple C., Whinston B., p. 19-53, Springer Verlag Berlin Heidelberg, 1987.

[BODART,93], Bodart F., Pigneur Y., *"Conception assistée des systèmes d'information"*, Masson, Paris, 1993.

[BORLAND,95], *"Local InterBase User's Guide"*, Borland, 1995.

[CHEN,76], Chen Y.-S., *"The Entity-Relationship Model : Toward a Unified View of Data"*, in ACM Transactions on Data Base Systems, p. 9-36, vol. 1, n°1, 1976.

[CHEN,88], Chen Y.-S., *"An Entity-Relationship Approach to Decision Support and Expert Systems"*, in Decision Support Systems, p. 225-234, vol. 4, June 1988.

[DBMAIN,95], *"DB-MAIN Tutorial : Introduction to Database Design"*, Institut d'Informatique, FUNDP, Namur, 1995.

[DECHOW,88], Dechow E.L., *"Entity-Relationship Approach and Decision Support"*, Elsevier Science Publisher B.V., 1988.

[DEJESUS,95], Dejesus E., *"Dimensions of Data"*, in Byte, p. 139-148, April 1995.

[ENGLEBERT,95], Englebert V., *"Voyager 2, Reference Manual"*, Institut d'Informatique, FUNDP, Namur, 1995.

[FICHEFET,82], Fichefet J., *"Eléments de programmation linéaire"*, Notes provisoires, Deuxième version corrigée, Institut d'Informatique, FUNDP, Namur, 1982.

[FICHEFET,93], Fichefet J., *"Théorie des graphes"*, Notes de cours, Institut d'Informatique, FUNDP, Namur, 1993.

[GEOFFRION,91], Geoffrion A. M., *"The SML Language for Structured Modeling : Level 1 and 2"*, Spec. report, University of California, LA, July, 1991.

[HAINAUT,86], Hainaut J.-L., *"Méthodes + Programmes : Conception assistée des applications informatique 2. Conception de la base de données"*, Masson, Presse Universitaires de Namur, 1986.

[HAINAUT,94a], Hainaut J.-L, *"Bases de données et modèles de calcul"*, InterEditions, Paris, 1994.

[HAINAUT,94b], Hainaut J.-L. *"Cours de base de données : Conception d'une base de données"*, Notes provisoires, Institut d'Informatique, FUNDP, Namur, 1994.

[HICK,91], Hick J.-M., Fortemps J., *"Contribution à une méthodologie de développement d'applications d'aide à la décision"*, Mémoire de maîtrise, Institut d'Informatique, FUNDP, Namur, 1991.

[HALPIN,95], Halpin T., *"Conceptual Schema and Relational Database Design"*, Prentice Hall, 1995.

[HUDSON,89], Hudson S., King R., *"Cactis : A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System"*, in ACM Transactions on Data Base Systems, p. 291-321, vol. 14, n°3, September 1989.

[ISAKOWITZ, 95], Isakowitz T., Shimon S., Henry C. L. , *"Toward a Logical/Physical Theory of Spreadsheet Modeling"*, in ACM Transactions on Information Systems, p. 1-37, vol. 13, n° 1, January 1995.

[KARAGIANNIS,87], Karagiannis D., Schneider H.-J., *"Data and knowledge-base management system for decision support"*, in Decision Support Systems : Theory and Application, Holsapple C., Whinston B., p. 19-53, Springer Verlag Berlin Heidelberg, 1987.

[KONOPASEK,84], Konopasek M., Jayaraman S., *"The TK!Solver Book"*, Osborne/McGraw Hill, 1984.

[LAZIMY,89], Lazimy R., *"E^2R model and object-oriented representation for data management, process modeling and decision support"*, in Proc. of the 8th Intern. Conf. on the ER Approach, Toronto, October 1989, North-Holland, p. 129-151, 1990.

[LECHARLIER,80], Lecharlier B., *"Spécification des procédures de l'interpréteur du langage LSD 80"*, Institut d'Informatique, FUNDP, Namur, 1980.

[LOUCOPOULOS,91], Loucopoulos D., Theodoulodis B., Pantazis D., *"Business Rules Modeling : Conceptual Modeling and Object Oriented Specification"*, in Object Oriented Approaches on Information Systems, Elsevier Science Publisher B.V., p. 323-342, 1991.

[MORIARTY,95], Moriarty T., *"Modeling Data Warehouses"*, Enterprise View, p. 61-65, August 1995.

[OMP,87], *"OMP Optimisation"*, Beyers & Partners, Brasschaat, 1987.

[RAUH,94], Rauh O., Stickel E., *"Modeling Deductive Information Systems Using ERM^{ded}"*, in Proceedings of the fourth Annual Workshop on Information Technologies Systems, WITS, 1994.

[SCHIFF,88], Schiff J., *"The design of a knowledge Based Economic Modeling Tool (EMT) Prototype"*, DS-3, p. 229-240, 1988.

[SOON,91], Soon T.S., Ackley D., Carasik R.P., , Tryon D.C., Tson E.S., Tsur S., Zaniolo C., *"System Analysis for deductive Database Environments : an Enhanced Role for Aggregate Entities"*, 1991.

[ULLMAN,88], Ullman J.D., *"Principles of Database and Knowledge-Base Systems"*, vol. 1, Computer Science Press, 1988.

[WELDON,95], Weldon J.-L., *"Managing multidimensional data : Harnessing the power"*, in Database programming & design, p. 24-33, August 1995.

[VANDERLANS,93], van der Lans R. F., *"Introduction to SQL"*, Addison-Wesley, 1993.

Annexes

Annexe I : Exemple de schéma EA déductif

Cette annexe propose un exemple de schéma EA déductif. Nous nous sommes efforcés d'y inclure un grand nombre d'attributs dérivables afin d'illustrer toutes les possibilités du langage d'expression des formules de définition. Nous commençons par décrire sommairement le domaine d'application modélisé. Nous décrivons ensuite le schéma EA de base. Nous présentons finalement le schéma EA déductif ainsi que les formules de définition des attributs dérivables qui s'y trouvent.

I.1 Contexte

On se place dans le cadre d'une chaîne de magasins X. X possède des clients qui lui passent des commandes. Chaque commande fait référence à des produits en une certaine quantité. X vend des produits dans plusieurs régions. Chaque client de X habite une région. Chaque produit vendu par X est fourni par un ou plusieurs fournisseurs. Les fournisseurs de X vendent leurs produits dans un certain nombre de régions.

I.2 Schéma EA de base

X possède une base de données de production modélisée par le schéma EA de base suivant :

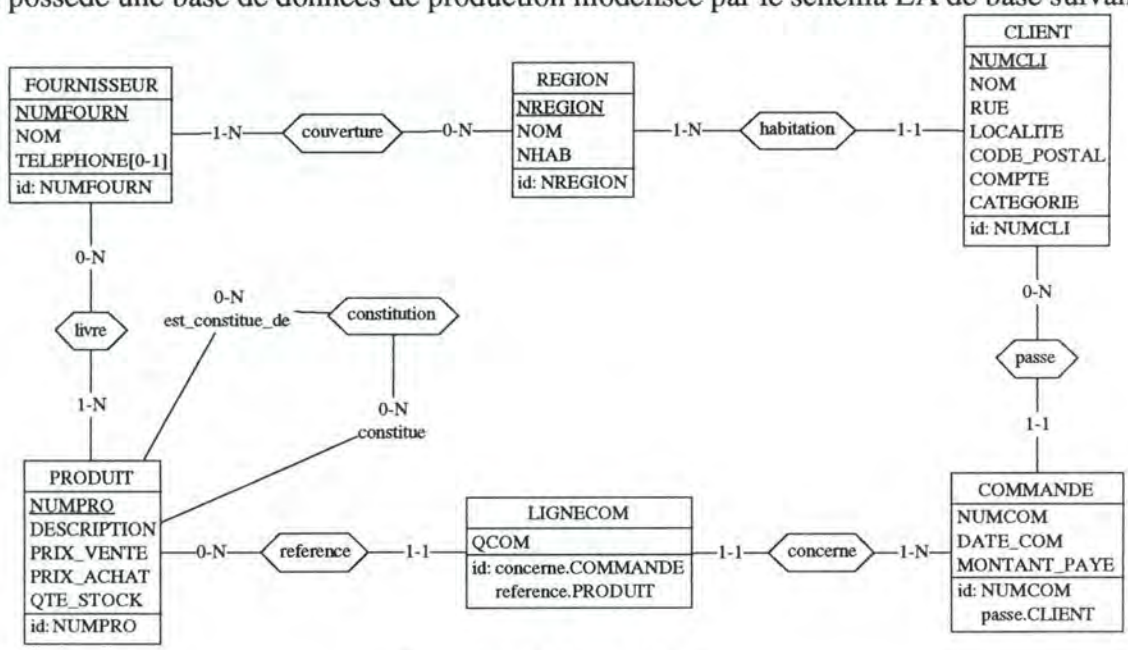


Figure I-1 : Schéma EA de base

Décrivons brièvement les différents composants de ce schéma.

Type d'entités CLIENT

Une entité `CLIENT` représente un client de l'entreprise. Un client est une personne ayant passé au moins une commande ou sur le point d'en passer une. Il est caractérisé par un numéro de client (`NUMCLI`), un nom (`NOM`), une rue (`RUE`), une localité (`LOCALITE`), un code postal (`CODE_POSTAL`), le montant qu'il possède sur son compte (`COMPTE`) et la catégorie à laquelle il appartient (`CATEGORIE`) : 1 s'il s'agit d'un cadre élevé, 2 s'il s'agit d'un cadre, 3 s'il s'agit d'un employé, 4 s'il s'agit d'un ouvrier ou 5 s'il s'agit d'un chômeur.

Un client habite une région (CLIENT est associé à REGION via habitation) et il passe un certain nombre de commandes (CLIENT est associé à COMMANDE via passe). Un client est identifié par son numéro (NUMCLI) unique.

Type d'entités COMMANDE

Une entité COMMANDE représente une commande qui a été passée par un client de l'entreprise. Une commande est caractérisée par un numéro de commande (NUMCOM), la date à laquelle elle a été passée (DATE_COM) et le montant que le client a déjà versé pour son payement (MONTANT_PAYE).

Une commande est passée par un client (COMMANDE est associé à CLIENT via passe) et possède un certain nombre de lignes de commande (COMMANDE est associé à LIGNECOM via concerne). Une commande est identifiée par le client qui l'a passée et par son numéro de commande (NUMCOM) unique parmi les commandes de ce client.

Type d'entités PRODUIT

Une entité PRODUIT représente un produit proposé par l'entreprise à sa clientèle. Un produit est caractérisé par un numéro (NUMPRO), une description (DESCRIPTION), un prix d'achat (PRIX_ACHAT), un prix de vente (PRIX_VENTE) et une quantité disponible en stock (QTE_STOCK).

Un produit peut être livré par plusieurs fournisseurs (PRODUIT est associé à FOURNISSEUR via livre) et il est désigné dans un certain nombre de lignes de commande (PRODUIT est associé à LIGNECOM via reference). Un produit est constitué d'autres produits et, inversement, un produit entre dans la constitution d'autres produits (PRODUIT est associé à lui-même via constitution). Un produit est identifié par son numéro (NUMPRO) unique.

Type d'entités LIGNECOM

Une entité LIGNECOM représente une ligne de commande désignant un certain produit p dans une certaine commande c. Une ligne de commande est caractérisée par la quantité du produit p commandée dans c (QCOM).

Une ligne de commande fait référence à un produit (LIGNECOM est associé à PRODUIT via reference) et est associée à une commande (LIGNECOM est associé à COMMANDE via concerne). Une ligne de commande est identifiée par la commande et par le produit auxquels elle est associée.

Type d'entités FOURNISSEUR

Une entité FOURNISSEUR représente un fournisseur livrant des produits vendus par l'entreprise. Un fournisseur est caractérisé par un numéro (NUMFOUR), un nom (NOM) et un numéro de téléphone éventuel (TELEPHONE).

Un fournisseur livre un certain nombre de produits à l'entreprise (FOURNISSEUR est associé à PRODUIT via livre) et il vend ses produits dans un certain nombre de régions (FOURNISSEUR est associé à REGION via couverture). Un fournisseur est identifié par son numéro (NUMFOUR) unique.

Type d'entités REGION

Une entité REGION représente une région géographique (par exemple une province). Une région est caractérisée par un numéro (NREGION), un nom (NOM) et le nombre d'habitants qui y vivent (NHAB).

Une région est couverte par des fournisseurs (REGION est associé à FOURNISSEUR via couverture) et elle est habitée par des clients (REGION est associé à CLIENT via habitation). Une REGION est identifiée par son numéro (NREGION) unique.

I.3 Schéma EA déductif

A partir des données stockées dans la base de données de production, les gestionnaires de X aimeraient dériver un certain nombre de nouvelles informations nécessaires à la gestion et au contrôle de l'entreprise. Au niveau conceptuel, on spécifie ces besoins en informations dérivées dans le schéma EA par le concept d'attribut dérivable.

Les gestionnaires voudraient notamment établir, pour un certain nombre de périodes, des statistiques relatives aux clients et aux régions. Ces informations ne savent pas être modélisées dans le schéma EA tel qu'il se présente dans la figure I-1 : dans ce schéma, on ne connaît pas le concept de période. Avant de construire le schéma EA déductif, il est nécessaire d'ajouter trois nouveaux types d'entités dans le schéma EA de la figure I-1 :

- PERIODE : une entité PERIODE représente une période de temps. Elle est identifiée par une date de début (DATE_DEBUT) et une date de fin (DATE_FIN);
- STATISTIQUE_REG : une entité STATISTIQUE_REG représente un ensemble de statistiques relatives à une région sur une période. Elle ne possède pas d'attribut de base : tous ses attributs sont des attributs dérivables. Elle est identifiée par la région et la période auxquelles elle se rapporte;
- STATISTIQUE_CLT : une entité STATISTIQUE_CLT représente un ensemble de statistiques relatives à un client sur une période. Elle ne possède pas d'attribut de base : tous ses attributs sont des attributs dérivables. Elle est identifiée par le client et la période auxquels elle se rapporte;

La figure I-2 représente le schéma EA déductif correspondant au schéma de base de la figure I-1 enrichi des trois nouveaux types d'entités et des attributs dérivables. Les attributs dérivables y sont indiqués en italique.

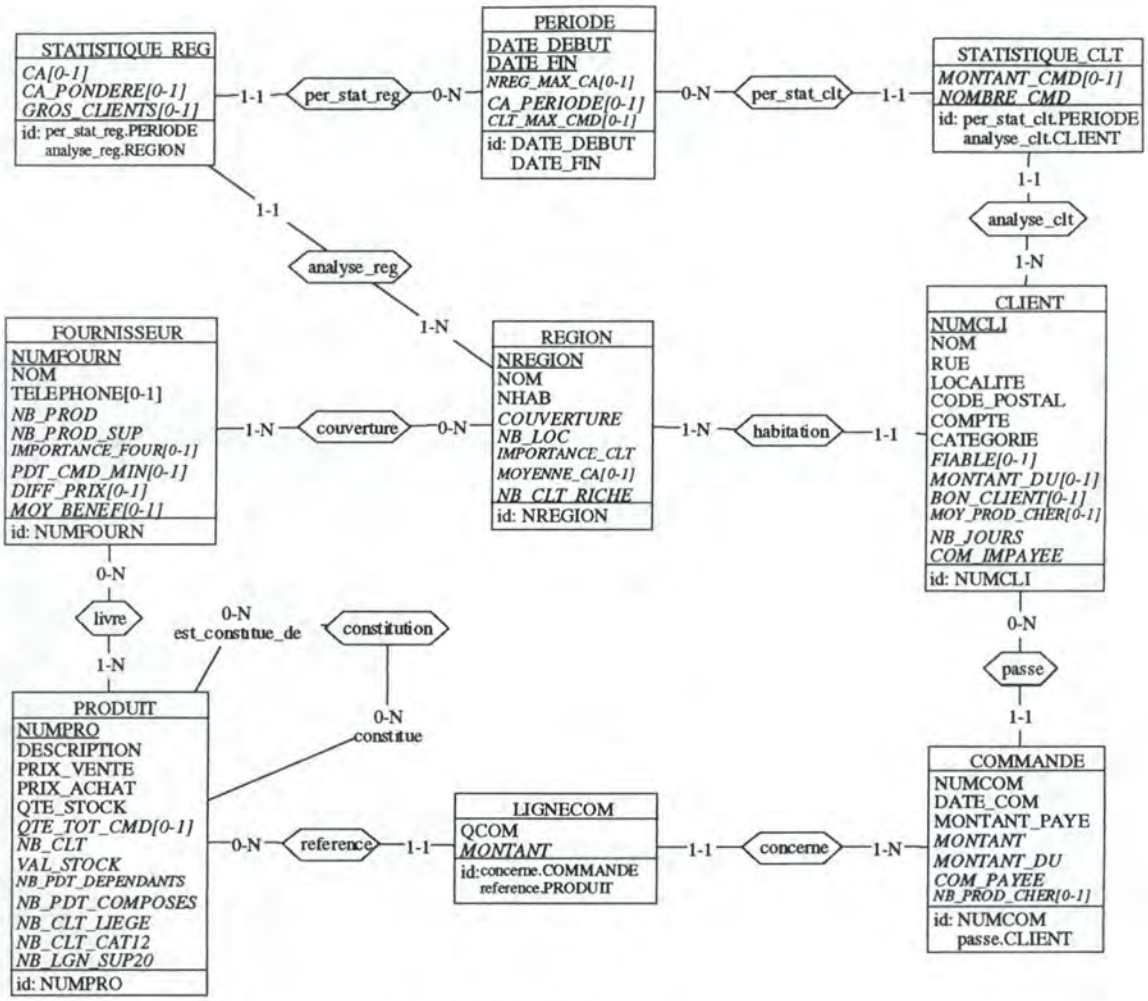


Figure I-2 : Schéma EA déductif

Décrivons à présent chaque attribut dérivable en donnant sa formule de définition.

Type d'entités LIGNECOM

Attribut MONTANT (Type : Numérique)
Représente le montant de la ligne de commande. Ce montant est obtenu en multipliant le prix du produit désigné par la ligne de commande par la quantité commandée de ce produit.

```
DEF= $.QCOM * PRODUIT(reference:$).PRIX_VENTE
```

Type d'entités COMMANDE

Attribut MONTANT (Type : Numérique)
Représente le montant total de la commande [40].

```
DEF= Somme( I=LIGNECOM(concerne:$);
            I.QCOM * PRODUIT(reference:I).PRIX_VENTE)
```

Attribut MONTANT_DU (Type : Numérique)
Représente le montant qu'il reste à payer par le client pour cette commande.

```
DEF= $.MONTANT - $.MONTANT_PAYE
```

[40] Il serait plus logique, dans cette formule de définition, d'utiliser l'attribut dérivable `MONTANT` de `LIGNECOM` au lieu de recalculer explicitement le montant de chaque ligne de commande.

Attribut COM_PAYEE (Type : Booléen)
Représente le fait que la commande est entièrement payée (COM_PAYEE=Vrai) ou non (COM_PAYEE=Faux). Ce résultat est obtenu en comparant le montant total de la commande avec la somme déjà versée par le client.

```
DEF= $.MONTANT = $.MONTANT_PAYE
```

Attribut NB_PROD_CHER (Type : Numérique)
Représente le nombre de produits chers contenus dans la commande. Un produit est cher si son prix est supérieur à la moyenne des prix des produits.

```
DEF= Somme(I=LIGNECOM(concerne:$ and reference:PRODUIT(:PRIX_VENTE >
                                        Moyenne(I=PRODUIT;I.PRIX_VENTE)));
           I.QCOM)
```

Type d'entités CLIENT

Attribut FIABLE (Type : Booléen)
Représente le fait qu'un client est fiable (FIABLE=Vrai) ou non (FIABLE=Faux). Un client est fiable s'il a, sur son compte, suffisamment d'argent que pour pouvoir payer ce qu'il doit à l'entreprise (total des montants dus sur chaque commande) et s'il appartient à la catégorie 1 ou 2.

```
DEF=  $.COMPTE > Somme(I=COMMANDE(passe:$); I.MONTANT_DU)
      and ($.CATEGORIE = 1 or $.CATEGORIE=2)
```

Attribut MONTANT_DU (Type : Numérique)
Représente le montant dû par le client à l'entreprise (somme des montants dus sur chaque commande)[41].

```
DEF= Somme(I=COMMANDE(passe:$);I.MONTANT - I.MONTANT_PAYE)
```

Attribut BON_CLIENT (Type : Booléen)
Représente le fait que le client est un bon client de l'entreprise (BON_CLIENT=Vrai) ou non (BON_CLIENT=Faux). Un bon client est un client qui a commandé des produits pour plus de 100.000 FB.

```
DEF= Somme( I=LIGNECOM(concerne:COMMANDE(passe:$) ) ;
            I.MONTANT)  > 100000
```

Attribut MOY_PROD_CHER (Type : Numérique)
Représente le nombre de produits chers (prix supérieur à la moyenne des prix des produits) achetés en moyenne par commande passée par le client.

```
DEF= Moyenne(I=COMMANDE(passe:$);I.NB_PROD_CHER)
```

Attribut NB_JOURS (Type : Numérique)
Représente le nombre de jours différents où le client a passé des commandes en 1995.

```
DEF=Nombreval(I=COMMANDE(     passe:$
                          and :DATE_COM>=950101
                          and :DATE_COM<=951231);I.DATE_COM)
```

Attribut COM_IMPAYEE (Type : Numérique)
Représente le nombre de commandes que le client n'a pas payées entièrement.

```
DEF= Nombre(COMMANDE(passe:$ and :MONTANT_DU > 0))
```

---

[41] Il serait plus logique, dans cette formule de définition, d'utiliser l'attribut dérivable MONTANT_DU de COMMANDE au lieu de recalculer explicitement le montant dû sur chaque commande.

Type d'entités PRODUIT

Attribut QTE_TOT_CMD (Type : Numérique)
Représente le nombre total de fois que ce produit a été commandé.
```
DEF= Somme (I=LIGNECOM(reference:$);I.QCOM)
```

Attribut NB_CLT (Type : Numérique)
Représente le nombre de clients distincts qui ont acheté ce produit.
```
DEF= Nombre(CLIENT(passe:COMMANDE(concerne:LIGNECOM(reference:$))))
```

Attribut VAL_STOCK (Type : Numérique)
Représente la valeur du stock pour ce produit dépréciée de 10%.
```
DEF= ($.PRIX_ACHAT * $.QTE_STOCK) * (9/10)
```

Attribut NB_PDT_DEPENDANTS (Type : Numérique)
Représente le nombre de produits que ce produit constitue (on utilise le rôle `constitue`).
```
DEF= Nombre(constitue(constitution:$))
```

Attribut NB_PDT_COMPOSES (Type : Numérique)
Représente le nombre de produits qui sont constitués de ce produit (on utilise le rôle `est_constitue_de`).
```
DEF= Nombre(est_constitue_de(constitution:$))
```

Attribut NB_CLT_LIEGE (Type : Numérique)
Représente le nombre de clients liégeois (clients habitant la région "PROVINCE DE LIEGE") qui ont acheté ce produit.
```
DEF= Nombre (CLIENT(:NUMCLI in
           CLIENT(habitation:REGION(:NOM="PROVINCE DE LIEGE")).NUMCLI
             and passe:COMMANDE(concerne:LIGNECOM(reference:$))))
```

Attribut NB_CLT_CAT12 (Type : Numérique)
Représente le nombre de clients de catégorie 1 ou 2 qui ont acheté ce produit.
```
DEF= Nombre(CLIENT(     :CATEGORIE in {1,2}
                  and passe:COMMANDE(concerne:LIGNECOM(reference:$))))
```

Attribut NB_LGN_SUP20 (Type : Numérique)
Représente le nombre de fois que ce produit a été commandé à 20 exemplaires ou plus.
```
DEF= Nombre(LIGNECOM(reference:$ and :QCOM>=20))
```

Type d'entités FOURNISSEUR

Attribut NB_PROD (Type : Numérique)
Représente le nombre de produits du fournisseur dont le prix de vente est supérieur a 1000 et qui ont été vendus au moins 100 fois.
```
DEF=Nombre (PRODUIT(     livre:$
                  and :PRIX_VENTE>1000
                  and :QTE_TOT_CMD >100))
```

Attribut NB_PROD_SUP (Type : Numérique)
Représente le nombre de produits du fournisseur dont le prix est supérieur à 200 et qui ont été vendus au moins une fois à 20 exemplaires ou plus.

```
DEF= Nombre (PRODUIT(    livre:$
                    and :PRIX_VENTE>=200
                    and :NB_LGN_SUP20 > 0))
```

Attribut IMPORTANCE_FOUR (Type : Numérique)
Représente le pourcentage de la quantité totale des produits vendus par l'entreprise qui concerne des produits livrés par ce fournisseur.

```
DEF=  Somme(I=PRODUIT(livre:$) ; I.QTE_TOT_CMD) /
      Somme(I=PRODUIT ; I.QTE_TOT_CMD) *100
```

Attribut PDT_CMD_MIN (Type : Numérique)
Représente le numéro du produit de ce fournisseur le moins vendu. Si plusieurs produits satisfont à ce critère, on prend le produit dont le numéro est le plus petit.

```
DEF=Min(I=PRODUIT(:QTE_TOT_CMD=Min(I=PRODUIT(livre:$);I.QTE_TOT_CMD)
                  and livre:$);
        I.NUMPRO)
```

Attribut DIFF_PRIX (Type : Numérique)
Représente la différence entre le prix maximal et minimal des produits livrés par ce fournisseur.

```
DEF= Max(I=PRODUIT(livre:$);I.PRIX_VENTE) -
     Min(I=PRODUIT(livre:$);I.PRIX_VENTE)
```

Attribut MOY_BENEF (Type : Numérique)
Représente la différence moyenne entre le prix d'achat et le prix de vente des produits de ce fournisseur.

```
DEF= Moyenne(I=PRODUIT(livre:$);I.PRIX_VENTE - I.PRIX_ACHAT)
```

Type d'entités STATISTIQUE_CLT

Rappelons qu'une entité de STATISTIQUE_CLT contient des statistiques de vente relatives à un client pour une période.

Attribut MONTANT_CMD (Type : Numérique)
Représente la somme pour laquelle le client a commandé des produits au cours de la période.

```
DEF= Somme(I=COMMANDE(passe:CLIENT(analyse_clt:$)
                    and :DATE_COM<=PERIODE(per_stat_clt:$).DATE_FIN
                    and :DATE_COM>=PERIODE(per_stat_clt:$).DATE_DEBUT);
           I.MONTANT)
```

Attribut NOMBRE_CMD (Type : Numérique)
Représente le nombre de commandes passées par le client au cours de la période.

```
DEF=Nombre(COMMANDE(    passe:CLIENT(analyse_clt:$)
                   and :DATE_COM<=PERIODE(per_stat_clt:$).DATE_FIN
                   and :DATE_COM>=PERIODE(per_stat_clt:$).DATE_DEBUT))
```

Type d'entités STATISTIQUE_REG

Rappelons qu'une entité de STATISTIQUE_REG contient des statistiques de vente relatives à une région pour une période.

Attribut CA (Type : Numérique)
Représente le montant total des commandes passées dans une région pour une période (équivaut au chiffre d'affaires d'une région au cours d'une période).

```
DEF= Somme(I=COMMANDE(passe:CLIENT(habitation:REGION(analyse_reg:$))
                  and :DATE_COM <= PERIODE(per_stat_reg:$).DATE_FIN
                  and :DATE_COM >= PERIODE(per_stat_reg:$).DATE_DEBUT);
           I.MONTANT)
```

Attribut CA_PONDERE (Type : Numérique)
Représente le chiffre d'affaires pondéré d'une région au cours d'une période. On pondère le CA de la région par rapport au nombre d'habitants de celle-ci : on ramène le CA de la région comme s'il y avait, dans cette région, autant d'habitants que dans la région qui en comporte le plus. Le seul CA qui ne change pas est donc celui de la région la plus peuplée.

```
DEF=$.CA*(Max(I=REGION;I.NHAB) / REGION(analyse_reg:$).NHAB)
```

Attribut GROS_CLIENTS (Type : Numérique)
Représente le pourcentage du CA total d'une période et d'une région couvert par les gros clients (clients qui ont commandé pour plus de 100.000 FB).

```
DEF= Somme(I=COMMANDE(passe:CLIENT(   :BON_CLIENT=True
                                  and habitation:REGION(analyse_reg:$))
                  and :DATE_COM<= PERIODE(per_stat_reg:$).DATE_FIN
                  and :DATE_COM>= PERIODE(per_stat_reg:$).DATE_DEBUT);
           I.MONTANT) / $.CA * 100
```

Type d'entités REGION

Attribut COUVERTURE (Type : Numérique)
Représente le pourcentage de la population de la région qui est cliente de l'entreprise.

```
DEF= Nombre(CLIENT(habitation:$))  / $.NHAB * 100
```

Attribut NB_LOC (Type : Numérique)
Représente le nombre de localités distinctes (de la région) où habitent des clients de l'entreprise.

```
DEF= NombreVal(I=CLIENT(habitation:$);I.LOCALITE)
```

Attribut IMPORTANCE_CLT (Type : Numérique)
Représente le pourcentage des clients de l'entreprise qui habitent dans la région.

```
DEF= Nombre(CLIENT(habitation:$)) / Nombre(CLIENT) * 100
```

Attribut MOYENNE_CA (Type : Numérique)
Représente la moyenne du chiffre d'affaires pour la région.

```
DEF=Moyenne(I=STATISTIQUE_REG(analyse_reg:$);I.CA)
```

Attribut NB_CLT_RICHE (Type : Numérique)
Représente le nombre de clients riches qui habitent la région. Un client est considéré comme "riche" s'il possède plus de 100.000 F sur son compte.

```
DEF= Nombre(CLIENT(     habitation:$
                  and :NUMCLI in CLIENT(:COMPTE>100000).NUMCLI))
```

Type d'entités PERIODE

Attribut NREG_MAX_CA (Type : Char)
Représente le nom de la région qui a fait le plus gros chiffre d'affaires au cours de cette période. Si plusieurs régions satisfont à ce critère, on prend la région dont le nom est alphabétiquement inférieur aux autres.

```
DEF= Min(I=REGION(analyse_reg:STATISTIQUE_REG(per_stat_reg:$  and
                  :CA=Max(I=STATISTIQUE_REG(per_stat_reg:$);I.CA)));
          I.NOM)
```

Attribut CA_PERIODE (Type : Numérique)
Représente le montant total des commandes passées au cours de la période.

```
DEF=Somme(I=COMMANDE(       :DATE_COM<=$.DATE_FIN
                        and :DATE_COM>=$.DATE_DEBUT);
           I.MONTANT)
```

Attribut CLT_MAX_CMD (Type : Char)
Représente le nom du client qui a commandé pour le montant le plus important au cours de la période. Si plusieurs clients satisfont à ce critère, on prend celui dont le nom est alphabétiquement inférieur à tous les autres.

```
DEF= Min(I=CLIENT(analyse_clt:STATISTIQUE_CLT(:MONTANT_CMD=
                  Max(I=STATISTIQUE_CLT(per_stat_clt:$);I.MONTANT_CMD)
                    and per_stat_clt:$));
          I.NOM)
```

Annexe II : Etude de cas

Dans cette annexe, nous étudions le cas concret d'un problème d'aide à la décision modélisé à l'aide du modèle EA déductif. Ce cas s'inspire du travail réalisé au cours du stage de fin d'études. Nous commençons par décrire le domaine d'application. Nous définissons ensuite le modèle EA de base modélisant la base de données de production. Nous présentons finalement le modèle EA déductif permettant de dériver les informations nécessaires à la prise de décision à partir des informations opérationnelles modélisées dans le schéma EA de base.

II.1 Contexte

Nous nous situons dans le contexte d'une entreprise multinationale solidement ancrée dans les pays européens. Cette entreprise commercialise de nombreux produits de grande consommation.

Afin de réduire de manière significative les coûts, le Top-Management a décidé de diminuer le nombre de variantes différentes d'un même type de produits offert par l'entreprise à sa clientèle sans pour autant réduire le choix de celle-ci. On s'est rendu compte qu'un même type de produits était mis sur le marché sous de nombreuses formes (variantes) différentes : large gamme de poids différents, langues différentes (un emballage pour chaque langue), promotions différentes (chaque promotion réalisée sur le produit donne naissance à un emballage spécifique et donc à une nouvelle variante du produit), etc. On désigne une variante particulière d'un type de produits en tant que *Stock Keeping Unit* (SKU).

Le produit *LEIRA Ultra*, par exemple, possède plus de 30 SKUs différents : emballage de 2 Kg, 3 Kg, 3.5 Kg, 5 Kg, etc. avec explications en français, allemand, anglais, italien, etc. Ce produit fait, de plus, l'objet de nombreuses campagnes de promotion : certains SKUs sont accompagnés d'un bon de remise, d'autres de gadgets divers, etc.

Cette trop grande diversité de SKUs rend le marché de l'entreprise très complexe et est à l'origine d'un surcoût important : chaque SKU doit être géré séparément au niveau des stocks, des campagnes marketing, des commandes clients, des lignes de production, etc. Le programme mis sur pied par le Top-Management (*SKU Reduction Program*) a pour objectif de baisser considérablement la complexité du marché en diminuant la diversité inutile des SKUs. Il permettra, par conséquent, d'éliminer les coûts provenant de cette complexité. Les filiales de tous les pays et tous les départements sont invités à faire de sérieux efforts pour mettre en pratique cette diminution.

Dans le cadre de ce large programme, le département informatique est chargé de fournir chaque mois le nombre de SKUs en vente dans les différents pays pour chaque marque et secteur d'activités de l'entreprise. Ces rapports doivent servir au Top-Management pour analyser l'évolution du nombre de SKUs au niveau des pays et des départements et pour s'assurer que la tendance est à la baisse. Ils doivent aussi servir, à un niveau plus opérationnel, à déterminer quels SKUs doivent être éliminés ou, au contraire, conservés.

II.2 Schéma EA de base

L'entreprise dispose d'une base de données dont le schéma conceptuel est représenté par la figure II-1. Il s'agit d'une base de données de production contenant des informations cruciales pour l'entreprise. Passons en revue tous les composants du schéma conceptuel et donnons en la signification.

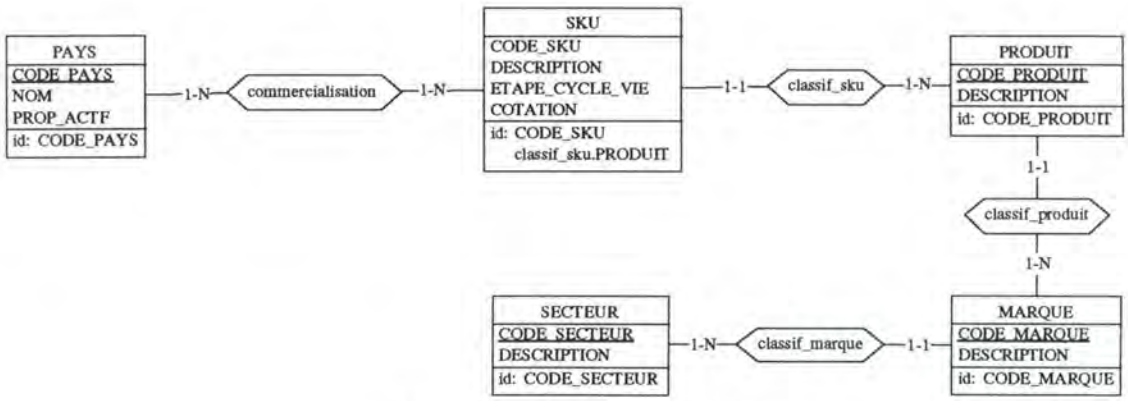


Figure II-1 : Schéma EA de base

Une entité `SKU` représente un SKU commercialisé par l'entreprise. Un SKU est une variante spécifique d'un produit donné. Un SKU est associé au produit dont il est une variante par le type d'associations `classif_sku` (exemple : *LEIRA Ultra* 3Kg avec emballage en italien et un coupon de remise est associé au produit *LEIRA Ultra*). Il est associé aussi à l'ensemble des pays dans lesquels il est commercialisé par le type d'associations `commercialisation` (exemple : *LEIRA Ultra* 3Kg avec emballage en allemand est commercialisé en Allemagne, en Autriche, en Belgique et au Luxembourg).

Un SKU est caractérisé par un code (attribut `CODE_SKU`) qui l'identifie parmi les SKUs appartenant au même produit. Il est caractérisé aussi par une description (attribut `DESCRIPTION`).

Chaque SKU possède un cycle de vie :

- au moment où on commence à le produire et qu'il est mis sur le marché, on dit qu'il est **actif**;
- quand on cesse définitivement de le produire mais qu'on continue à le vendre (pour épuiser les stocks), on dit qu'il est **résiduel**;
- quand, finalement, il n'est plus produit ni vendu, on dit qu'il est **mort**.

L'étape du cycle de vie dans lequel un SKU se trouve est indiquée par l'attribut `ETAPE_CYCLE_VIE`. Celui-ci possède la valeur `ACTF` si le SKU est actif, `RESD` s'il est résiduel et `MORT` s'il est mort.

Au sein de l'entreprise, on attribue à chaque SKU une cote sur 10 établie en fonction de différents critères : prix de revient, espace nécessaire au stockage, volume de vente, bénéfice, etc. Une formule mathématique complexe permet, selon la valeur de chaque critère, de fixer la cote du SKU. Plus celle-ci se rapproche de 10, plus le SKU est intéressant pour l'entreprise, le SKU idéal étant celui qui occupe un très petit espace de stockage, qui a un très petit prix de revient, un énorme volume de vente et un énorme bénéfice. La cote du SKU est placée dans l'attribut `COTATION`.

Une entité PRODUIT représente un produit de l'entreprise. Un produit est caractérisé par un code (attribut CODE_PRODUIT) et une description (attribut DESCRIPTION). Le code d'un produit identifie celui-ci au sein de tous les produits de l'entreprise. Un produit est associé à la marque à laquelle il appartient par le type d'associations classif_produit (exemple : *LEIRA Ultra* appartient à la marque *LEIRA*).

Le type d'entités MARQUE regroupe toutes les marques dont l'entreprise est titulaire. Chaque marque est caractérisée par un code (attribut CODE_MARQUE) et une description (attribut DESCRIPTION). Le code d'une marque identifie celle-ci dans l'ensemble de toutes les marques de l'entreprise.

Une entité SECTEUR représente un secteur d'activités où l'entreprise est présente. Chaque marque est associée au secteur dans lequel elle est active par le type d'associations classif_marque (exemple : *LEIRA* appartient au secteur *Poudres à lessiver*). Chaque secteur est caractérisé par un code (attribut CODE_SECTEUR) et une description (attribut DESCRIPTION). Le code d'un secteur identifie celui-ci dans l'ensemble de tous les secteurs de l'entreprise.

Une entité PAYS représente un pays dans lequel l'entreprise commercialise certains SKUs. Chaque pays est identifié par un code (attribut CODE_PAYS) et par un nom (attribut NOM). Un pays est associé à tous les SKUs qui sont commercialisés dans celui-ci par le type d'associations commercialisation.

II.3 Schéma EA déductif

A partir des informations présentes dans la base de données de production, on aimerait maintenant dériver l'information nécessaire aux gestionnaires pour, d'une part, suivre l'évolution du nombre de SKUs et, d'autre part, déterminer où se trouvent les SKUs candidats à une suppression. Au niveau du schéma conceptuel, on définit un attribut dérivable pour chaque requête du management. En enrichissant le schéma EA de base par des éléments dérivables, on transforme celui-ci en un schéma EA déductif.

Le schéma EA déductif est représenté par la figure II-2. Les attributs dérivables y sont indiqués en italique.

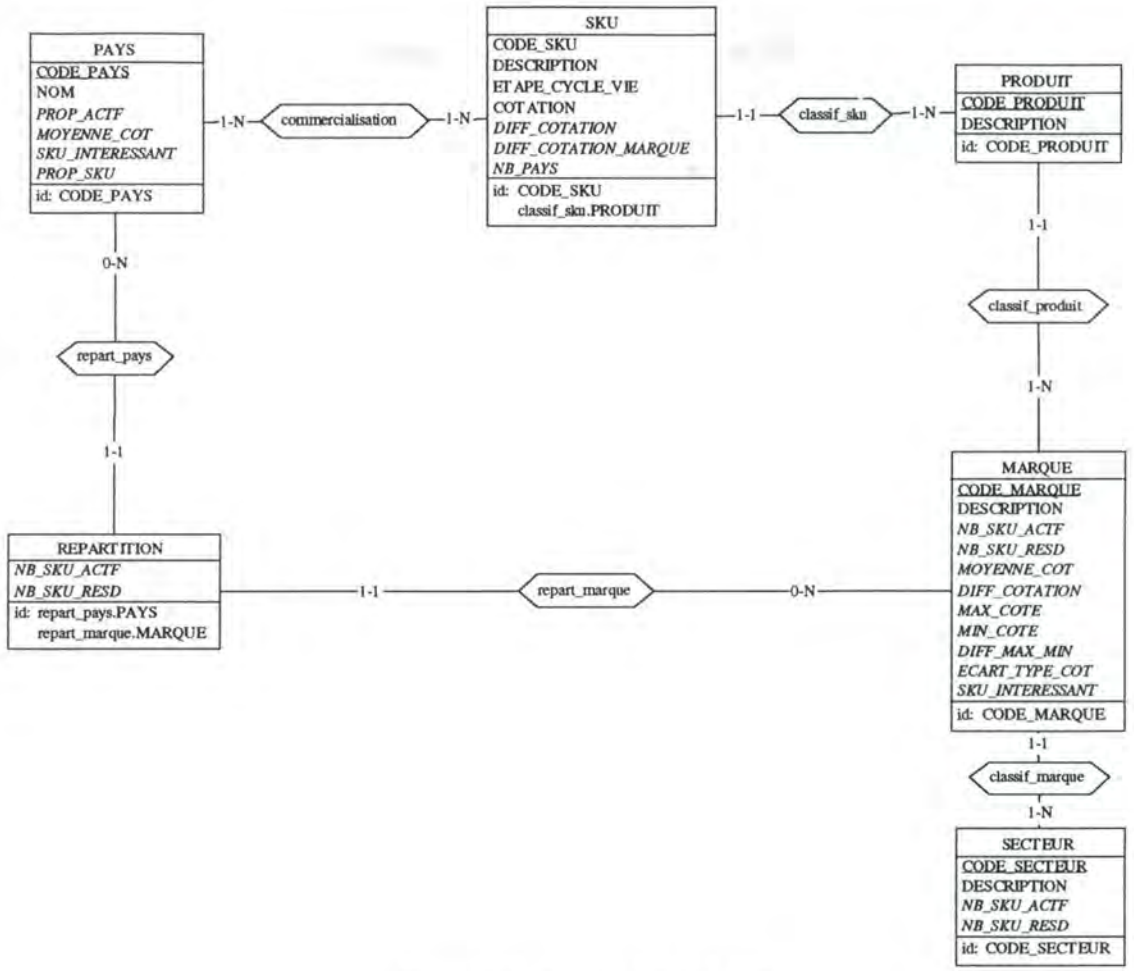


Figure II-2 : Schéma EA déductif

Au niveau global, on désire connaître le nombre de SKUs actifs et résiduels pour chaque marque de l'entreprise, indépendamment des pays dans lesquels ceux-ci sont commercialisés.

Pour compter le nombre de SKUs actifs de chaque marque, on définit l'attribut dérivable `NB_SKU_ACTF` de `MARQUE` par la formule de définition suivante :

```
DEF= Nombre(SKU(:ETAPE_CYCLE_VIE = "ACTF"
                and classif_sku:PRODUIT(classif_produit:$)))
```

Pour compter le nombre de SKUs résiduels de chaque marque, on définit l'attribut dérivable `NB_SKU_RESD` de `MARQUE` par la formule de définition suivante :

```
DEF= Nombre(SKU(:ETAPE_CYCLE_VIE = "RESD"
                and classif_sku:PRODUIT(classif_produit:$)))
```

Dans le même ordre d'idée, on désire connaître le nombre de SKUs actifs et résiduels au niveau de chaque secteur. Pour obtenir ces mesures, il n'est plus nécessaire de compter le nombre de SKUs au niveau du type d'entités `SKU` puisqu'on dispose déjà de ce nombre pour chaque marque : il suffit d'additionner le nombre de SKUs pour chaque marque d'un même secteur afin obtenir le nombre de SKUs du secteur.

Pour les SKUs actifs d'un secteur, on définit l'attribut dérivable NB_SKU_ACTF dans le type d'entités SECTEUR :

```
DEF= Somme(I=MARQUE(classif_marque:$);I.NB_SKU_ACTF)
```

Pour les SKUs résiduels d'un secteur, on définit l'attribut dérivable NB_SKU_RESD dans le type d'entités SECTEUR :

```
DEF= Somme(I=MARQUE(classif_marque:$);I.NB_SKU_RESD)
```

On désire connaître aussi, pour chaque marque, le nombre de SKUs commercialisés dans chaque pays. Pour enregistrer des informations au sujet d'une marque et d'un pays, il est nécessaire de construire un nouveau type d'entités que nous appelons REPARTITION. Il est associé à PAYS par le type d'associations repart_pays et à MARQUE par repart_marque. Une entité de REPARTITION est identifiée par le pays et la marque auxquels elle se rapporte.

Pour le calcul du nombre de SKUs actifs d'une marque donnée commercialisés dans un pays donné, on définit l'attribut dérivable NB_SKU_ACTIF dans le type d'entités REPARTTION :

```
DEF= Nombre (
          SKU(:ETAPE_CYCLE_VIE="ACTF" and
              commercialisation:PAYS(repart_pays:$)and
              classif_sku:PRODUIT(classif_produit:
                                  MARQUE(repart_marque:$))))
```

Pour le calcul du nombre de SKUs résiduels d'une marque donnée commercialisés dans un pays donné, on définit l'attribut dérivable NB_SKU_RESD dans le type d'entités REPARTTION :

```
DEF= Nombre (
          SKU(:ETAPE_CYCLE_VIE="RESD" and
              commercialisation:PAYS(repart_pays:$)and
              classif_sku:PRODUIT(classif_produit:
                                  MARQUE(repart_marque:$))))
```

Au niveau de chaque pays, on aimerait connaître la proportion (en pourcentage) de SKUs actifs commercialisés dans ce pays par rapport au nombre total de SKUs commercialisés dans ce pays. On définit l'attribut PROP_ACTF de PAYS :

```
DEF=  Nombre(SKU(commercialisation:$ and :ETAPE_CYCLE_VIE="ACTF")) /
      Nombre(SKU(commercialisation:$)) * 100
```

Les attributs dérivables qui viennent d'être définis permettent de suivre, mois par mois, l'évolution du nombre de SKUs. On définit maintenant des attributs dérivables qui permettent de déterminer à quels endroits il s'agit de faire des efforts pour réduire le nombre de SKUs. En d'autres mots, on essaye de déterminer quels sont les marques et les pays qui comprennent ou commercialisent les SKUs les moins avantageux pour l'entreprise : c'est au niveau de ces pays et de ces marques que les éliminations de SKUs superflus vont se concentrer.

Pour déterminer si un SKU est avantageux ou non pour notre entreprise, on se base sur sa cote (attribut COTATION de SKU). Dans un premier temps, on aimerait comparer la cote de chaque SKU à la cote des autres SKUs de l'entreprise. Pour ce faire, on définit l'attribut

dérivable DIFF_COTATION qui représente la différence entre la cotation d'un SKU et la moyenne des cotations des SKUs de l'entreprise. Il est défini comme suit :

```
DEF= $.COTATION - Moyenne(I=SKU;I.COTATION)
```

Si la valeur de DIFF_COTATION est positive pour un SKU, cela signifie que sa cote est supérieure à la moyenne des cotes et le SKU se trouve parmi les SKUs les plus intéressants pour l'entreprise. Si, par contre, elle est négative, sa cote est inférieure à la moyenne.

On aimerait maintenant déterminer la cote moyenne des SKUs appartenant à une même marque. On définit pour ce faire l'attribut dérivable MOYENNE_COT de MARQUE par la formule de définition suivante :

```
DEF= Moyenne(I=SKU(classif_sku:PRODUIT(classif_produit:$));I.COTATION)
```

Au sein d'une même marque, on aimerait comparer la cote de chaque SKU à la cote des autres SKUs de la marque. Pour effectuer cette comparaison, on mesure la différence entre la cote de ce SKU et la cote moyenne des SKUs de la même marque. On définit l'attribut dérivable DIFF_COTATION_MARQUE de SKU :

```
DEF=  $.COTATION -
      MARQUE(classif_produit:PRODUIT(classif_sku:$)).MOYENNE_COT
```

Afin de repérer les marques contenant les SKUs les moins rentables, on aimerait comparer la moyenne des cotes d'une marque (attribut MOYENNE_COT de MARQUE) à la moyenne des cotes de toutes les marques. On définit l'attribut dérivable DIFF_COTATION de MARQUE comme suit :

```
DEF= $.MOYENNE_COT - Moyenne(I=MARQUE;I.MOYENNE_COT)
```

Au sein d'une même marque, on aimerait avoir une idée de la répartition des cotes des SKUs de la marque. Pour ce faire, on détermine pour chaque marque la cote la plus haute des SKUs de cette marque, la cote la plus basse ainsi que leur différence. On définit, dans MARQUE, les attributs dérivables MAX_COTE, MIN_COTE et DIFF_MAX_MIN :

- MAX_COTE est défini par
  `DEF= Max(I=SKU(classif_sku:PRODUIT(classif_produit:$));I.COTATION)`

- MIN_COTE est défini par
  `DEF= Min(I=SKU(classif_sku:PRODUIT(classif_produit:$));I.COTATION)`

- DIFF_MAX_MIN est défini par
  `DEF= $.MAX_COTE - $.MIN_COTE`

On aimerait savoir aussi dans quelle mesure les cotes des SKUs de cette marque s'écarte de la moyenne des cotes de la marque. Cela revient à calculer l'écart type des cotes des SKUs d'une même marque (moyenne des écarts par rapport à la moyenne). On définit l'attribut dérivable ECART_TYPE_COT de MARQUE par la formule :

```
DEF=Moyenne(I=SKU(classif_sku:PRODUIT(classif_produit:$));
            I.DIFF_COTATION)
```

Pour chaque marque, on aimerait connaître la proportion (en pourcentage) de SKUs de cette marque jugés intéressants par rapport au nombre total de SKUs de la marque. Un SKU est jugé intéressant si sa cote dépasse 7. On définit l'attribut SKU_INTERESSANT de MARQUE :

```
DEF=
Nombre(SKU(classif_sku:PRODUIT(classif_produit:$) and :COTATION >=7)) /
Nombre(SKU(classif_sku:PRODUIT(classif_produit:$))) * 100
```

Les attributs dérivables qu'on vient de définir permettent, d'une part, d'analyser les performances des SKUs au sein de chaque marque et, d'autre part, de comparer les marques entre elles afin de déterminer les moins intéressantes. Nous nous tournons à présent vers les pays.

Au niveau d'un pays, on aimerait connaître la moyenne des cotes des SKUs commercialisés dans ce pays ainsi que la proportion de SKUs intéressants (cote >= 7). On définit dans PAYS les attributs dérivables MOYENNE_COT et SKU_INTERESSANT :

- l'attribut MOYENNE_COT de PAYS est défini par :
  ```
  DEF= Moyenne(I=SKU(commercialisation:$);I.COTATION)
  ```
- l'attribut SKU_INTERESSANT de PAYS est défini par :
  ```
  DEF= Nombre(SKU(commercialisation:$ and :COTATION>=7)) /
       Nombre(SKU(commercialisation:$)) * 100
  ```

On aimerait connaître aussi pour chaque pays la proportion (en pourcentage) de SKUs commercialisés dans ce pays par rapport au nombre total de SKUs de l'entreprise. On définit l'attribut dérivable PROP_SKU de PAYS :

```
DEF= Nombre(SKU(commercialisation:$)) / Nombre(SKU) * 100
```

Pour finir, on aimerait connaître pour chaque SKU le nombre de pays dans lesquels celui-ci est commercialisé. Il s'agit là d'un critère qui peut être important pour déterminer si le SKU en question doit être conservé ou non. On définit l'attribut NB_PAYS de SKU :

```
DEF= Nombre(PAYS(commercialisation:$))
```

Annexe III : Syntaxe BNF complète du langage

Cette annexe résume la syntaxe BNF complète du langage d'expression des formules de définition du modèle EA déductif.

Le tableau suivant rappelle les conventions BNF utilisées :

Symbole BNF	Signification
e	Symbole terminal du langage
e	Symbole non terminal du langage
::=	Définit un symbole non terminal du langage
[...]	Elément facultatif
{...}	Elément qui peut être répété de 0 à N fois
...|...|...	Choix d'un élément parmi N

```
ConstNum::= [Signe]Chiffre{Chiffre}
Chiffre ::= 1|2|3|4|5|6|7|8|9|0
Signe   ::= -

ConstAlphanum::= "ChaîneCar"
ChaîneCar ::=   {Caractère}
Caractère ::=   Lettre | Chiffre | CarSpécial
Lettre ::=      LettreMaj | LettreMin
LettreMaj ::=   A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|
                V|W|X|Y|Z
LettreMin ::=   a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|
                v|w|x|y|z
CarSpécial ::=  !| |#|$|%|&|'|(|)|*|+|,|-|.|/|:|;|<|=|>|?|@|[|
                \|]|^|_|{|}|`|~

ConstBool ::=   TRUE | FALSE

TEntité ::=     NomElement
TAssociation::= NomElement
Attribut ::=    NomElement
NomElement ::=  Lettre {Lettre | Chiffre} | _ }

Fd ::=          DEF=Expr

Expr ::=        ExprSimple | ExprArithm | ExprLog

ExprSimple ::=  Constante | Attr | AppelFct
Constante ::=   ConstBool |     ConstNum | ConstAlphanum

ExprArithm ::=  Terme | Terme OpAdditif ExprArithm
Terme ::=       Facteur | Facteur OpMult Terme
Facteur ::=     ExprSimple | - Facteur |(ExprArithm)
OpAdditif ::=   - | +
OpMult ::=      * | /

ExprLog ::=     Conjonction or ExprLog | Conjonction
Conjonction::=  Négation and Conjonction | Négation
Négation ::=    PropAtom | not Negation | (ExprLog)
PropAtom ::=    ExprArithm | ExprArithm OpComp ExprArithm
OpComp ::=      <|<=|>|>=|=|<>
```

```
Attr ::=          EnsEnt.Attribut

EnsEnt ::=        TEntité | $  | TEntité(Csel) | I

Csel ::=          CselConj | CselConj or Csel
CselConj ::=      CselNeg  | CselNeg  and CselConj
CselNeg ::=       Cass     | not CselNeg | (Csel)

Cass ::=          CassEntité | CassAttribut
CassEntité ::=    (TAssociation:EnsEnt)
CassAttribut::=   (:Attribut Capp)

Capp ::=          Rel EnsVal
Rel ::=           OpComp | OpEnsembliste
OpEnsembliste::=in | not in

EnsVal ::=        ExprSimple | {ExprSimple{,ExprSimple}}

AppelFct      ::= FctSomme  | FctMin      | FctMax |
                  FctNombre       | FctMoyenne | FctNombreVal
FctSomme      ::= Somme(I=EnsEnt;Expr_I)
FctMin        ::= Min(I=EnsEnt;Expr_I)
FctMax        ::= Max(I=EnsEnt;Expr_I)
FctMoyenne    ::= Moyenne(I=EnsEnt;Expr_I)
FctNombreVal  ::= NombreVal(I=EnsEnt;Expr_I)
FctNombre     ::= Nombre(EnsEnt)
```

Annexe IV: Exemple complet de génération de scripts

Cette annexe propose un exemple complet de génération de scripts SQL à partir d'un schéma EA déductif. Nous commençons par décrire le schéma EA déductif en donnant les formules de définition des attributs dérivables qui y figurent. Nous présentons ensuite le schéma logique correspondant à ce schéma EA déductif. Nous décrivons finalement les scripts SQL issus du processus de génération complet : script 1, script 2 et script 3. Nous nous sommes efforcés d'inclure dans le schéma EA déductif la plus grande variété possible de formules de définition afin d'illustrer le maximum de mécanismes de génération présentés au chapitre 8.

IV.1 Schéma EA déductif

Le schéma EA déductif sur lequel se base notre exemple est celui de la figure IV-1. Les attributs dérivables y sont indiqués en italique. Le schéma EA de base correspondant à ce schéma EA déductif étant d'une compréhension relativement intuitive, nous ne nous attarderons pas à décrire les types d'entités, types d'associations et attributs de base[42]. Nous nous limiterons à la description des attributs dérivables.

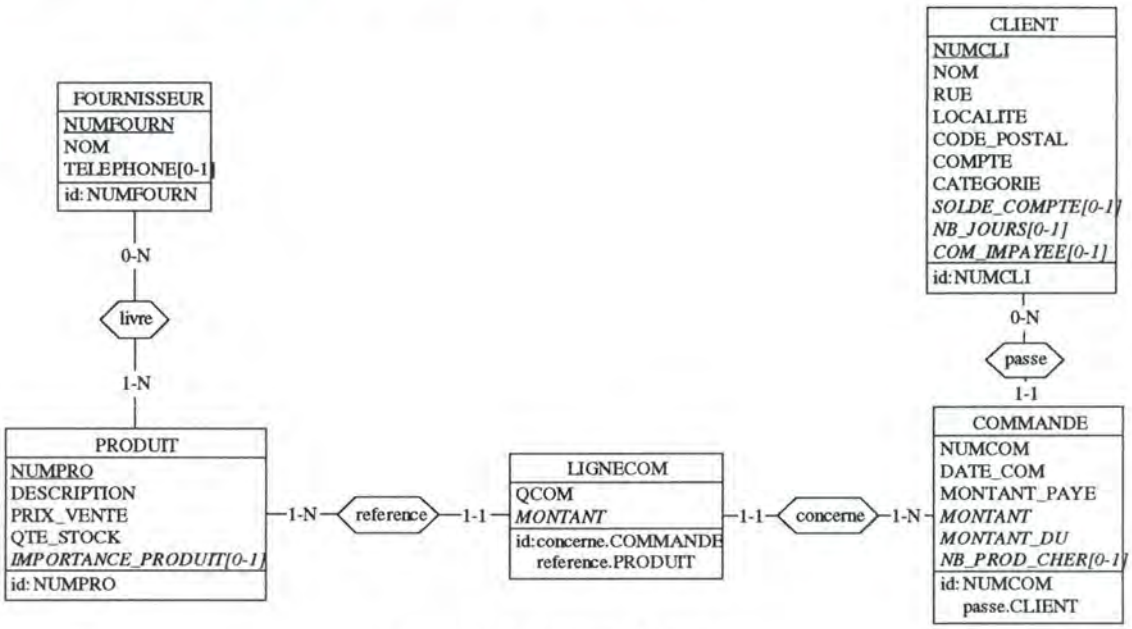


Figure IV-1 : Schéma EA déductif

Type d'entités LIGNECOM

Attribut MONTANT (Type : Numérique)
Représente le montant de la ligne de commande. Ce montant est obtenu en multipliant le prix du produit désigné par la ligne de commande par la quantité commandée de ce produit.

```
DEF= $.QCOM * PRODUIT(reference:$).PRIX_VENTE
```

[42] Le schéma EA de base correspondant à notre exemple est un sous-ensemble du schéma EA de base présenté dans l'annexe I (figure I-1) : le lecteur y trouvera une description détaillée des éléments de base du schéma.

Type d'entités COMMANDE

Attribut MONTANT (Type : Numérique)
Représente le montant total de la commande. Ce résultat est obtenu en additionnant le montant de toutes les lignes de commande qui la constituent.

```
DEF= Somme (I=LIGNECOM(concerne:$);I.MONTANT)
```

Attribut MONTANT_DU (Type : Numérique)
Représente le montant dû sur cette commande.

```
DEF= $.MONTANT - $.MONTANT_PAYE
```

Attribut NB_PROD_CHER (Type : Numérique)
Représente la quantité maximale de produits chers de la commande. Un produit est considéré comme cher si son prix est supérieur à la moyenne des prix des produits. Une commande contient un certain nombre de produits chers (peut-être aucun). Chacun de ces produits est commandé en une certaine quantité. L'attribut NB_PROD_CHER représente, parmi celles-ci, la quantité maximale.

```
DEF= Max(I=LIGNECOM(     concerne:$
                    and reference:PRODUIT(:PRIX_VENTE>
                                Moyenne(I=PRODUIT;I.PRIX_VENTE)));
         I.QCOM)
```

Type d'entités CLIENT

Attribut SOLDE_COMPTE (Type : Numérique)
Représente le solde du compte tel qu'il se présenterait si le client payait le montant qu'il doit à l'entreprise (somme des montants dus sur chaque commande).

```
DEF= $.COMPTE - Somme(I=COMMANDE(passe:$);I.MONTANT - I.MONTANT_PAYE)
```

Attribut NB_JOURS (Type : Numérique)
Représente le nombre de jours différents où le client a passé des commandes en 1995.

```
DEF=Nombreval(I=COMMANDE(     passe:$
                         and :DATE_COM>=950101
                         and :DATE_COM<=951231);I.DATE_COM)
```

Attribut COM_IMPAYEE (Type : Numérique)
Représente le nombre de commandes que le client n'a pas payées entièrement.

```
DEF= Nombre(COMMANDE(passe:$ and :MONTANT_DU > 0))
```

Type d'entités PRODUIT

Attribut IMPORTANCE_PRODUIT (Type : Numérique)
Représente la proportion (en pourcentage) de la quantité totale vendue d'un produit par rapport à la quantité totale vendue, tous les produits confondus.

```
DEF= Somme(I=LIGNECOM(reference:$);I.QCOM) /
     Somme(I=LIGNECOM;I.QCOM) *100
```

IV.2 Transformation du schéma EA déductif en schéma logique

Le schéma EA déductif de la figure IV-1 (schéma conceptuel) est transformé en un schéma EA logique. Lors de cette transformation, nous ignorons la présence des attributs dérivables. Le schéma EA logique résultant est celui de la figure IV-2.

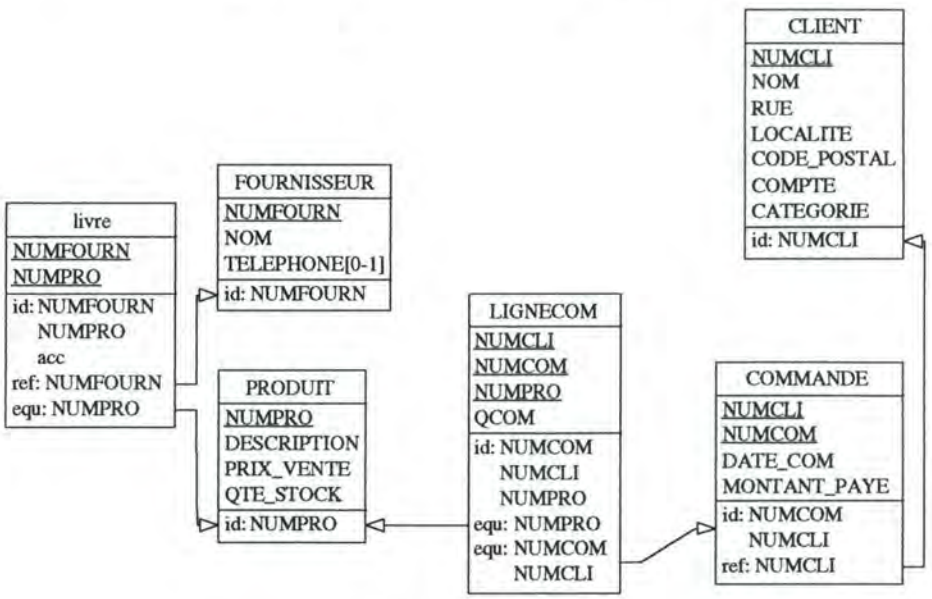


Figure IV-2 : Schéma logique

A partir du schéma logique, le passage au schéma physique correspondant est immédiat : les types d'entités sont transformés en tables et les attributs en colonnes.

IV.3 Génération de scripts SQL

A partir du schéma EA déductif et de son schéma logique (ou physique) correspondant, nous générons les 3 scripts SQL destinés à rendre le schéma EA déductif exécutable : script 1, script 2 et script 3. Les scripts présentés ici ont été générés automatiquement par le système que nous avons implémenté dans l'atelier DB-MAIN.

IV.3.1 Script 1

Le script 1 contient toutes les requêtes SQL nécessaires à la création des tables de base. Il est construit exclusivement à partir du schéma logique. Les tables de base étant créées une fois pour toutes, le script 1 ne devrait, normalement, être soumis qu'une seule fois sur la base de données.

Le script 1 est généré automatiquement par l'atelier DB-MAIN[43]. Contrairement aux scripts 2 et 3, le script 1 n'est pas construit pour pouvoir être exécuté par un SGBDR spécifique : il s'agit d'un script SQL standard. Certaines retouches sont, par conséquent, nécessaires pour le rendre conforme aux spécifications d'un SGBDR particulier.

[43] La génération de ce script est une fonctionnalité offerte par DB-MAIN.

```
/* ***************************
/* * Standard SQL generation *
/* ***************************

/* Database Section
/* ________________

create database DEMO.GDB;                    La base de données créée est nommée DEMO.GDB

/* Table Section
/* _____________

create table CLIENT (
      NUMCLI numeric(10) not null,
      NOM char(25) not null,
      RUE char(25) not null,
      LOCALITE char(15) not null,
      CODE_POSTAL numeric(4) not null,
      COMPTE numeric(5) not null,
      CATEGORIE numeric(1) not null,
      primary key (NUMCLI));

create table COMMANDE (
      NUMCLI numeric(10) not null,
      NUMCOM numeric(5) not null,
      DATE_COM numeric(6) not null,
      MONTANT_PAYE numeric(5) not null,
      primary key (NUMCOM, NUMCLI));

create table FOURNISSEUR (
      NUMFOURN numeric(5) not null,
      NOM char(25) not null,
      TELEPHONE char(10),
      primary key (NUMFOURN));

create table LIGNECOM (
      NUMCLI numeric(10) not null,
      NUMCOM numeric(5) not null,
      NUMPRO numeric(5) not null,
      QCOM numeric(5) not null,
      primary key (NUMCOM, NUMCLI, NUMPRO));

create table livre (
      NUMFOURN numeric(5) not null,
      NUMPRO numeric(5) not null,
      primary key (NUMFOURN, NUMPRO));

create table PRODUIT (
      NUMPRO numeric(5) not null,
      DESCRIPTION char(25) not null,
      PRIX_VENTE numeric(5) not null,
      QTE_STOCK numeric(5) not null,
      primary key (NUMPRO));
```

IV.3.2 Script 2

Le script 2 contient toutes les requêtes SQL nécessaires à la création des tables complémentaires. Il est construit à partir du schéma EA déductif. Les tables complémentaires étant créées une fois pour toutes, le script 2 ne devrait, normalement, être soumis qu'une seule fois sur la base de données.

Le script 2 est généré automatiquement par le système que nous avons intégré dans l'atelier DB-MAIN et est conçu pour être exécuté par le SGBDR *InterBase* de Borland.

```
/*****************************************************************
 CONNECT TO DATABASE
 *****************************************************************/
CONNECT DEMO.GDB USER JBE PASSWORD password;
```

On commence par se connecter à la base de données DEMO.GDB créée par le script 1 en fournissant l'identification de l'utilisateur et son mot de passe

```
/*****************************************************************
 TABLE CREATION
 *****************************************************************/

/*****************************************************************/
CREATE TABLE COMP_CLIENT(
                NUMCLI NUMERIC(10,0) NOT NULL,
                SOLDE_COMPTE NUMERIC(5,0) ,
                NB_JOURS NUMERIC(5,0) ,
                COM_IMPAYEE NUMERIC(5,0) ,
                PRIMARY KEY( NUMCLI ) );

/*****************************************************************/
CREATE TABLE COMP_COMMANDE(
                NUMCOM NUMERIC(5,0) NOT NULL,
                NUMCLI NUMERIC(10,0) NOT NULL,
                MONTANT NUMERIC(5,0) NOT NULL ,
                MONTANT_DU NUMERIC(5,0) NOT NULL ,
                NB_PROD_CHER NUMERIC(5,0) ,
                PRIMARY KEY( NUMCOM , NUMCLI ) );

/*****************************************************************/
CREATE TABLE COMP_LIGNECOM(
                NUMCOM NUMERIC(5,0) NOT NULL,
                NUMCLI NUMERIC(10,0) NOT NULL,
                NUMPRO NUMERIC(5,0) NOT NULL,
                MONTANT NUMERIC(5,0) NOT NULL ,
                PRIMARY KEY( NUMCOM , NUMCLI , NUMPRO ) );

/*****************************************************************/
CREATE TABLE COMP_PRODUIT(
                NUMPRO NUMERIC(5,0) NOT NULL,
                IMPORTANCE_PRODUIT NUMERIC(5,0) ,
                PRIMARY KEY( NUMPRO ) );

/*****************************************************************
 COMMIT WORK ON DATABASE
 *****************************************************************/
EXIT;
```

Le fait de terminer le script SQL par un EXIT permet de faire un COMMIT sur la base de données et ainsi de rendre permanentes les modifications réalisées par le script 2.

IV.3.3 Script 3

Le script 3 contient toutes les requêtes SQL nécessaires au calcul effectif des valeurs des attributs dérivables et au stockage de ces valeurs dans les tables complémentaires créées par le script 2. Il est construit à partir du schéma EA déductif et, plus particulièrement, à partir des formules de définition des attributs dérivables qu'il contient. Le script 3 peut être soumis sur la base de données à chaque fois qu'on désire rafraîchir les valeurs des attributs dérivables.

Le script 3 est généré automatiquement par le système que nous avons intégré dans l'atelier DB-MAIN et est conçu pour être exécuté par le SGBDR *InterBase* de Borland.

```
/*************************************************************************
 CONNECT TO DATABASE
 *************************************************************************/
CONNECT DEMO.GDB USER JBE PASSWORD password;
```

On commence par se connecter à la base de données DEMO.GDB créée par le script 1 en fournissant l'identification de l'utilisateur et son mot de passe

On crée maintenant toutes les vues et tables intermédiaires nécessaires au calcul des attributs dérivables et on stocke les valeurs de chaque attribut dérivable dans une table temporaire : requêtes SQL de niveau 1 et 2.

```
/*************************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE CLIENT.NB_JOURS
 *************************************************************************/

/*************************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande
Nombreval(I=COMMANDE(passe:$
and :DATE_COM>=950101
and :DATE_COM<=951231);I.DATE_COM)

```
CREATE VIEW CLIENT_NB_JOURS_1(NUMCLI,OPERANDE)
 AS
SELECT CLIENT_1.NUMCLI, COUNT(DISTINCT COMMANDE_1.DATE_COM)
FROM   CLIENT CLIENT_1, COMMANDE COMMANDE_1
WHERE  CLIENT_1.NUMCLI = COMMANDE_1.NUMCLI
AND    COMMANDE_1.DATE_COM >= 950101
AND    COMMANDE_1.DATE_COM <= 951231
GROUP BY CLIENT_1.NUMCLI;

/*************************************************************************/
CREATE TABLE CLIENT_NB_JOURS
      (     NUMCLI NUMERIC(10,0) NOT NULL,
            NB_JOURS NUMERIC(5,0),
            PRIMARY KEY( NUMCLI ) );
```

Création d'une table temporaire pour le stockage des valeurs de CLIENT.NB_JOURS

```
INSERT INTO CLIENT_NB_JOURS
SELECT CLIENT.NUMCLI,
       CLIENT_NB_JOURS_1.OPERANDE
FROM   CLIENT CLIENT, CLIENT_NB_JOURS_1 CLIENT_NB_JOURS_1
WHERE  CLIENT.NUMCLI = CLIENT_NB_JOURS_1.NUMCLI;
```

Calcul effectif de CLIENT.NB_JOURS et stockage des résultats dans la table temporaire

```
/*********************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE COMMANDE.NB_PROD_CHER
 *********************************************************************/

/*********************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande
Max(I=LIGNECOM(concerne:$ and reference:PRODUIT(:PRIX_VENTE>
Moyenne(I=PRODUIT;I.PRIX_VENTE)));
I.QCOM)

```
CREATE VIEW COMMANDE_NB_PROD_CHER_1(NUMCOM,NUMCLI,OPERANDE)
 AS
SELECT COMMANDE_1.NUMCOM, COMMANDE_1.NUMCLI, MAX(LIGNECOM_1.QCOM)
FROM   COMMANDE COMMANDE_1, LIGNECOM LIGNECOM_1, PRODUIT PRODUIT_1
WHERE  LIGNECOM_1.NUMCOM = COMMANDE_1.NUMCOM
AND    COMMANDE_1.NUMCLI = LIGNECOM_1.NUMCLI
AND    PRODUIT_1.NUMPRO = LIGNECOM_1.NUMPRO
AND    PRODUIT_1.PRIX_VENTE > ( SELECT    AVG(PRODUIT_1.PRIX_VENTE)
                                FROM      PRODUIT PRODUIT_1 )
GROUP BY COMMANDE_1.NUMCOM,COMMANDE_1.NUMCLI;

/*********************************************************************/
CREATE TABLE COMMANDE_NB_PROD_CHER
      (     NUMCOM NUMERIC(5,0) NOT NULL,
            NUMCLI NUMERIC(10,0) NOT NULL,
            NB_PROD_CHER NUMERIC(5,0),
            PRIMARY KEY( NUMCOM , NUMCLI ) );
```

Création d'une table temporaire pour le stockage des valeurs de l'attribut COMMANDE.NB_PROD_CHER

```
INSERT INTO COMMANDE_NB_PROD_CHER
SELECT COMMANDE.NUMCOM,
       COMMANDE.NUMCLI,
       COMMANDE_NB_PROD_CHER_1.OPERANDE
FROM   COMMANDE COMMANDE,
       COMMANDE_NB_PROD_CHER_1 COMMANDE_NB_PROD_CHER_1
WHERE  COMMANDE.NUMCOM = COMMANDE_NB_PROD_CHER_1.NUMCOM
AND    COMMANDE.NUMCLI = COMMANDE_NB_PROD_CHER_1.NUMCLI;
```

Calcul effectif de COMMANDE.NB_PROD_CHER et stockage des résultats dans la table temporaire

```
/*********************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE LIGNECOM.MONTANT
 *********************************************************************/

/*********************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande
PRODUIT(reference:$).PRIX_VENTE

```
CREATE VIEW LIGNECOM_MONTANT_1(NUMCOM,NUMCLI,NUMPRO,OPERANDE)
 AS
SELECT LIGNECOM_1.NUMCOM, LIGNECOM_1.NUMCLI, LIGNECOM_1.NUMPRO,
       PRODUIT_1.PRIX_VENTE
FROM   LIGNECOM LIGNECOM_1, PRODUIT PRODUIT_1
WHERE  LIGNECOM_1.NUMPRO = PRODUIT_1.NUMPRO;
```

```
/*********************************************************************/
CREATE TABLE LIGNECOM_MONTANT                    Création d'une table temporaire pour le
      (     NUMCOM NUMERIC(5,0) NOT NULL,        stockage des valeurs de l'attribut
            NUMCLI NUMERIC(10,0) NOT NULL,       dérivable LIGNECOM.MONTANT
            NUMPRO NUMERIC(5,0) NOT NULL,
            MONTANT NUMERIC(5,0) NOT NULL,
            PRIMARY KEY( NUMCOM , NUMCLI , NUMPRO ) );

INSERT INTO LIGNECOM_MONTANT                     Calcul effectif de
SELECT LIGNECOM.NUMCOM, LIGNECOM.NUMCLI,         LIGNECOM.MONTANT et stockage
       LIGNECOM.NUMPRO,                          des résultats dans la table temporaire
       (LIGNECOM.QCOM * LIGNECOM_MONTANT_1.OPERANDE)
FROM   LIGNECOM LIGNECOM , LIGNECOM_MONTANT_1 LIGNECOM_MONTANT_1
WHERE  LIGNECOM.NUMCOM = LIGNECOM_MONTANT_1.NUMCOM
AND    LIGNECOM.NUMCLI = LIGNECOM_MONTANT_1.NUMCLI
AND    LIGNECOM.NUMPRO = LIGNECOM_MONTANT_1.NUMPRO;

/*********************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE
 PRODUIT.IMPORTANCE_PRODUIT
 *********************************************************************/

/*********************************************************************/
Création d'une vue pour le calcul intermédiaire de l'opérande
      Somme(I=LIGNECOM(reference:$);I.QCOM)

CREATE VIEW PRODUIT_IMPORTANCE_PRODUIT_1(NUMPRO,OPERANDE)
 AS
SELECT PRODUIT_1.NUMPRO, SUM(LIGNECOM_1.QCOM)
FROM   PRODUIT PRODUIT_1, LIGNECOM LIGNECOM_1
WHERE  PRODUIT_1.NUMPRO = LIGNECOM_1.NUMPRO
GROUP  BY PRODUIT_1.NUMPRO;

/*********************************************************************/
Création d'une vue pour le calcul intermédiaire de l'opérande
      Somme(I=LIGNECOM;I.QCOM)

CREATE VIEW PRODUIT_IMPORTANCE_PRODUIT_2(NUMPRO,OPERANDE)
 AS
SELECT PRODUIT_1.NUMPRO, SUM(LIGNECOM_1.QCOM)
FROM   PRODUIT PRODUIT_1, LIGNECOM LIGNECOM_1
GROUP  BY PRODUIT_1.NUMPRO;

/*********************************************************************/
CREATE TABLE PRODUIT_IMPORTANCE_PRODUIT          Création d'une table temporaire pour
      (     NUMPRO NUMERIC(5,0) NOT NULL,        le stockage des valeurs de l'attribut
            IMPORTANCE_PRODUIT NUMERIC(5,0),     PRODUIT.IMPORTANCE_PRODUIT
            PRIMARY KEY( NUMPRO ) );

                                                 Calcul effectif de
INSERT INTO PRODUIT_IMPORTANCE_PRODUIT           PRODUIT.IMPORTANCE_PRODUIT
SELECT PRODUIT.NUMPRO,                           et stockage des résultats dans la table
       (PRODUIT_IMPORTANCE_PRODUIT_1.OPERANDE /                    temporaire
        PRODUIT_IMPORTANCE_PRODUIT_2.OPERANDE)  * 100)
FROM   PRODUIT PRODUIT,
       PRODUIT_IMPORTANCE_PRODUIT_1 PRODUIT_IMPORTANCE_PRODUIT_1,
       PRODUIT_IMPORTANCE_PRODUIT_2 PRODUIT_IMPORTANCE_PRODUIT_2
WHERE  PRODUIT.NUMPRO = PRODUIT_IMPORTANCE_PRODUIT_1.NUMPRO
AND    PRODUIT.NUMPRO = PRODUIT_IMPORTANCE_PRODUIT_2.NUMPRO;
```

```
/*************************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE COMMANDE.MONTANT
 *************************************************************************/

/*************************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande
Somme (I=LIGNECOM(concerne:$); I.MONTANT)

```
CREATE VIEW COMMANDE_MONTANT_1(NUMCOM,NUMCLI,OPERANDE)
 AS
SELECT COMMANDE_1.NUMCOM,
       COMMANDE_1.NUMCLI,
       SUM(LIGNECOM_MONTANT_1.MONTANT)
FROM   COMMANDE COMMANDE_1, LIGNECOM LIGNECOM_1,
       LIGNECOM_MONTANT LIGNECOM_MONTANT_1
WHERE  LIGNECOM_1.NUMCOM = LIGNECOM_MONTANT_1.NUMCOM
AND    LIGNECOM_1.NUMCLI = LIGNECOM_MONTANT_1.NUMCLI
AND    LIGNECOM_1.NUMPRO = LIGNECOM_MONTANT_1.NUMPRO
AND    LIGNECOM_1.NUMCOM = COMMANDE_1.NUMCOM
AND    COMMANDE_1.NUMCLI = LIGNECOM_1.NUMCLI
GROUP BY COMMANDE_1.NUMCOM,COMMANDE_1.NUMCLI;

/*************************************************************************/
CREATE TABLE COMMANDE_MONTANT                    Création d'une table temporaire
      (     NUMCOM NUMERIC(5,0) NOT NULL,        pour le stockage des valeurs de
            NUMCLI NUMERIC(10,0) NOT NULL,       COMMANDE.MONTANT
            MONTANT NUMERIC(5,0) NOT NULL,
            PRIMARY KEY( NUMCOM , NUMCLI ) );

INSERT INTO COMMANDE_MONTANT                     Calcul effectif de
SELECT COMMANDE.NUMCOM, COMMANDE.NUMCLI,         COMMANDE.MONTANT et stockage
       COMMANDE_MONTANT_1.OPERANDE               des résultats dans la table temporaire
FROM   COMMANDE COMMANDE, COMMANDE_MONTANT_1 COMMANDE_MONTANT_1
WHERE  COMMANDE.NUMCOM = COMMANDE_MONTANT_1.NUMCOM
AND    COMMANDE.NUMCLI = COMMANDE_MONTANT_1.NUMCLI;

/*************************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE CLIENT.SOLDE_COMPTE
 *************************************************************************/

/*************************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande
Somme(I=COMMANDE(passe:$);I.MONTANT - I.MONTANT_PAYE)

```
CREATE VIEW CLIENT_SOLDE_COMPTE_1(NUMCLI,OPERANDE)
 AS
SELECT CLIENT_1.NUMCLI,
       SUM(COMMANDE_MONTANT_1.MONTANT - COMMANDE_1.MONTANT_PAYE)
FROM   CLIENT CLIENT_1, COMMANDE COMMANDE_1,
       COMMANDE_MONTANT COMMANDE_MONTANT_1
WHERE  COMMANDE_1.NUMCOM = COMMANDE_MONTANT_1.NUMCOM
AND    COMMANDE_1.NUMCLI = COMMANDE_MONTANT_1.NUMCLI
AND    CLIENT_1.NUMCLI = COMMANDE_1.NUMCLI
GROUP BY CLIENT_1.NUMCLI;
```

```
/*************************************************************************/
CREATE TABLE CLIENT_SOLDE_COMPTE                    Création d'une table temporaire
      (     NUMCLI NUMERIC(10,0) NOT NULL,          pour le stockage des valeurs de
            SOLDE_COMPTE NUMERIC(5,0),              CLIENT.SOLDE_COMPTE
            PRIMARY KEY( NUMCLI ) );

                                                    Calcul effectif de
INSERT INTO CLIENT_SOLDE_COMPTE                     CLIENT.SOLDE_COMPTE et
                                                    stockage des résultats dans la table
SELECT CLIENT.NUMCLI,                               temporaire
       (CLIENT.COMPTE - CLIENT_SOLDE_COMPTE_1.OPERANDE)
FROM   CLIENT CLIENT,
       CLIENT_SOLDE_COMPTE_1 CLIENT_SOLDE_COMPTE_1
WHERE  CLIENT.NUMCLI = CLIENT_SOLDE_COMPTE_1.NUMCLI;

/*************************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE COMMANDE.MONTANT_DU
 *************************************************************************/

/*************************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande $.MONTANT
On crée une vue parce que COMMANDE.MONTANT est lui-même un attribut dérivable. Il est plus facile et plus général dans le processus de génération de créer une vue dans ce cas au lieu d'inclure l'opérande directement dans la requête de niveau 2.

```
CREATE VIEW COMMANDE_MONTANT_DU_1(NUMCOM,NUMCLI,OPERANDE)
 AS
SELECT COMMANDE_1.NUMCOM, COMMANDE_1.NUMCLI,
       COMMANDE_MONTANT_1.MONTANT
FROM   COMMANDE COMMANDE_1, COMMANDE_MONTANT COMMANDE_MONTANT_1
WHERE  COMMANDE_1.NUMCOM = COMMANDE_MONTANT_1.NUMCOM
AND    COMMANDE_1.NUMCLI = COMMANDE_MONTANT_1.NUMCLI;

/*************************************************************************/
CREATE TABLE COMMANDE_MONTANT_DU                    Création d'une table temporaire
      (     NUMCOM NUMERIC(5,0) NOT NULL,           pour le stockage des valeurs de
            NUMCLI NUMERIC(10,0) NOT NULL,          COMMANDE.MONTANT_DU
            MONTANT_DU NUMERIC(5,0) NOT NULL,
            PRIMARY KEY( NUMCOM , NUMCLI ) );
                                                    Calcul effectif de
INSERT INTO COMMANDE_MONTANT_DU                     COMMANDE.MONTANT_DU et
                                                    stockage des résultats dans la table
SELECT COMMANDE.NUMCOM, COMMANDE.NUMCLI,            temporaire
       (COMMANDE_MONTANT_DU_1.OPERANDE - COMMANDE.MONTANT_PAYE)
FROM   COMMANDE COMMANDE,
       COMMANDE_MONTANT_DU_1 COMMANDE_MONTANT_DU_1
WHERE  COMMANDE.NUMCOM = COMMANDE_MONTANT_DU_1.NUMCOM
AND    COMMANDE.NUMCLI = COMMANDE_MONTANT_DU_1.NUMCLI;
```

```
/*************************************************************************
 CREATE TEMPORARY TABLE AND VIEWS FOR ATTRIBUTE CLIENT.COM_IMPAYEE
 *************************************************************************/

/*************************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande
Nombre(COMMANDE(passe:$ and :MONTANT_DU > 0))
Cette vue sélectionne tous les couples distincts associant une entité de CLIENT à une entité de COMMANDE.

```
CREATE VIEW CLIENT_COM_IMPAYEE_1_1( NUMCLI, COMMANDE_NUMCOM,
                                    COMMANDE_NUMCLI)
 AS
SELECT DISTINCT CLIENT_1.NUMCLI, COMMANDE_1.NUMCOM,   COMMANDE_1.NUMCLI
FROM   CLIENT CLIENT_1, COMMANDE COMMANDE_1,
       COMMANDE_MONTANT_DU COMMANDE_MONTANT_DU_1
WHERE  CLIENT_1.NUMCLI = COMMANDE_1.NUMCLI
AND    COMMANDE_1.NUMCOM = COMMANDE_MONTANT_DU_1.NUMCOM
AND    COMMANDE_1.NUMCLI = COMMANDE_MONTANT_DU_1.NUMCLI
AND    COMMANDE_MONTANT_DU_1.MONTANT_DU > 0;

/*************************************************************************/
```

Création d'une vue pour le calcul intermédiaire de l'opérande
Nombre(COMMANDE(passe:$ and :MONTANT_DU > 0))
Cette vue se base sur la vue définie précédemment : elle se contente de compter le nombre de couples distincts pour chaque entité de CLIENT.

```
CREATE VIEW CLIENT_COM_IMPAYEE_1(NUMCLI,OPERANDE)
 AS
SELECT CLIENT_1.NUMCLI, COUNT(*)
FROM   CLIENT CLIENT_1, CLIENT_COM_IMPAYEE_1_1 CLIENT_COM_IMPAYEE_1_1
WHERE  CLIENT_1.NUMCLI = CLIENT_COM_IMPAYEE_1_1.NUMCLI
GROUP BY CLIENT_1.NUMCLI;

/*************************************************************************/
CREATE TABLE CLIENT_COM_IMPAYEE
      (     NUMCLI NUMERIC(10,0) NOT NULL,
            COM_IMPAYEE NUMERIC(5,0),
            PRIMARY KEY( NUMCLI ) );
```

Création d'une table temporaire pour le stockage des valeurs de CLIENT.COM_IMPAYEE

```
INSERT INTO CLIENT_COM_IMPAYEE
SELECT CLIENT.NUMCLI,CLIENT_COM_IMPAYEE_1.OPERANDE
FROM   CLIENT CLIENT,CLIENT_COM_IMPAYEE_1 CLIENT_COM_IMPAYEE_1
WHERE  CLIENT.NUMCLI = CLIENT_COM_IMPAYEE_1.NUMCLI;
```

Calcul effectif de CLIENT.COM_IMPAYEE et stockage des résultats dans la table temporaire

```
/*************************************************************************
 STORAGE OF DERIVED ATTRIBUTE VALUES INTO COMPLEMENTARY TABLES
 *************************************************************************/
```

Maintenant que toutes les valeurs des attributs dérivables sont présentes dans les tables temporaires, on les regroupe dans les tables complémentaires : requêtes SQL de niveau 3.

```
/*************************************************************************/
DELETE FROM COMP_CLIENT;      On efface le contenu courant de COMP_CLIENT
                              (table complémentaire de CLIENT)

                              On y introduit les nouvelles valeurs des attributs dérivables en faisant
INSERT INTO COMP_CLIENT       un jointure externe entre la table de base et chaque table temporaire
SELECT CLIENT.NUMCLI , CLIENT_SOLDE_COMPTE.SOLDE_COMPTE ,
       CLIENT_NB_JOURS.NB_JOURS , CLIENT_COM_IMPAYEE.COM_IMPAYEE
FROM CLIENT
     LEFT OUTER JOIN CLIENT_SOLDE_COMPTE
       on   CLIENT.NUMCLI = CLIENT_SOLDE_COMPTE.NUMCLI
     LEFT OUTER JOIN CLIENT_NB_JOURS
       on   CLIENT.NUMCLI = CLIENT_NB_JOURS.NUMCLI
     LEFT OUTER JOIN CLIENT_COM_IMPAYEE
       on   CLIENT.NUMCLI = CLIENT_COM_IMPAYEE.NUMCLI;

/*************************************************************************/
DELETE FROM COMP_COMMANDE;    On efface le contenu courant de COMP_COMMANDE
                              (table complémentaire de COMMANDE)

                              On y introduit les nouvelles valeurs des attributs dérivables en faisant
INSERT INTO COMP_COMMANDE     un jointure externe entre la table de base et chaque table temporaire
SELECT COMMANDE.NUMCOM , COMMANDE.NUMCLI , COMMANDE_MONTANT.MONTANT ,
       COMMANDE_MONTANT_DU.MONTANT_DU ,
       COMMANDE_NB_PROD_CHER.NB_PROD_CHER
FROM COMMANDE
     LEFT OUTER JOIN COMMANDE_MONTANT
       on   COMMANDE.NUMCOM = COMMANDE_MONTANT.NUMCOM
       and  COMMANDE.NUMCLI = COMMANDE_MONTANT.NUMCLI
     LEFT OUTER JOIN COMMANDE_MONTANT_DU
       on   COMMANDE.NUMCOM = COMMANDE_MONTANT_DU.NUMCOM
       and  COMMANDE.NUMCLI = COMMANDE_MONTANT_DU.NUMCLI
     LEFT OUTER JOIN COMMANDE_NB_PROD_CHER
       on   COMMANDE.NUMCOM = COMMANDE_NB_PROD_CHER.NUMCOM
       and  COMMANDE.NUMCLI = COMMANDE_NB_PROD_CHER.NUMCLI;

/*************************************************************************/
DELETE FROM COMP_LIGNECOM;    On efface le contenu courant de COMP_LIGNECOM
                              (table complémentaire de LIGNECOM)

                              On y introduit les nouvelles valeurs des attributs dérivables en faisant
INSERT INTO COMP_LIGNECOM     un jointure externe entre la table de base et chaque table temporaire
SELECT LIGNECOM.NUMCOM , LIGNECOM.NUMCLI , LIGNECOM.NUMPRO ,
       LIGNECOM_MONTANT.MONTANT
FROM LIGNECOM
     LEFT OUTER JOIN LIGNECOM_MONTANT
       on   LIGNECOM.NUMCOM = LIGNECOM_MONTANT.NUMCOM
       and  LIGNECOM.NUMCLI = LIGNECOM_MONTANT.NUMCLI
       and  LIGNECOM.NUMPRO = LIGNECOM_MONTANT.NUMPRO;
```

```
/*************************************************************************/
DELETE FROM COMP_PRODUIT;
```
On efface le contenu courant de COMP_PRODUIT (table complémentaire de PRODUIT)

On y introduit les nouvelles valeurs des attributs dérivables en faisant un jointure externe entre la table de base et chaque table temporaire

```
INSERT INTO COMP_PRODUIT
SELECT PRODUIT.NUMPRO , PRODUIT_IMPORTANCE_PRODUIT.IMPORTANCE_PRODUIT
FROM PRODUIT
     LEFT OUTER JOIN PRODUIT_IMPORTANCE_PRODUIT
       on    PRODUIT.NUMPRO = PRODUIT_IMPORTANCE_PRODUIT.NUMPRO;

COMMIT;
```
Un COMMIT est nécessaire à ce stade pour pouvoir, ensuite, détruire les vues et tables temporaires

```
/*************************************************************************
 DESTRUCTION OF TEMPORARY TABLES AND VIEWS
 *************************************************************************/
```
Maintenant que toutes les valeurs des attributs dérivables se trouvent dans les tables complémentaires, on peut détruire toutes les vues et tables intermédiaires de niveau 1 et 2 puisque celles-ci sont devenues inutiles.

```
DROP TABLE CLIENT_COM_IMPAYEE;
DROP VIEW CLIENT_COM_IMPAYEE_1;
DROP VIEW CLIENT_COM_IMPAYEE_1_1;
DROP TABLE COMMANDE_MONTANT_DU;
DROP VIEW COMMANDE_MONTANT_DU_1;
DROP TABLE CLIENT_SOLDE_COMPTE;
DROP VIEW CLIENT_SOLDE_COMPTE_1;
DROP TABLE COMMANDE_MONTANT;
DROP VIEW COMMANDE_MONTANT_1;
DROP TABLE PRODUIT_IMPORTANCE_PRODUIT;
DROP VIEW PRODUIT_IMPORTANCE_PRODUIT_2;
DROP VIEW PRODUIT_IMPORTANCE_PRODUIT_1;
DROP TABLE LIGNECOM_MONTANT;
DROP VIEW LIGNECOM_MONTANT_1;
DROP TABLE COMMANDE_NB_PROD_CHER;
DROP VIEW COMMANDE_NB_PROD_CHER_1;
DROP TABLE CLIENT_NB_JOURS;
DROP VIEW CLIENT_NB_JOURS_1;

/*************************************************************************
 COMMIT WORK ON DATABASE
 *************************************************************************/
EXIT;
```
On termine le script 3 en rendant permanentes toutes les modifications réalisées par celui-ci dans la base de données.

Annexe V : Application de l'algorithme de calcul des cardinalités

Cette annexe présente un exemple d'application du processus de calcul des cardinalités d'une expression de désignation d'un attribut (cfr. *6.1*). Nous illustrons l'algorithme présenté en *6.1.5* en l'appliquant sur trois expressions de désignation d'un attribut.

Nous nous basons sur le schéma EA déductif suivant (les attributs dérivables sont indiqués en italique) :

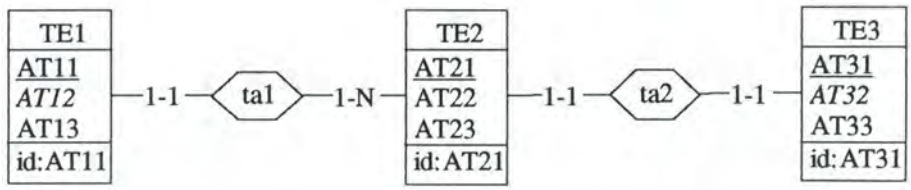


Figure V-1 : Schéma EA déductif

Les formules de définition des attributs dérivables se présentent comme suit :

- `TE1.AT12` est défini par `DEF= $.AT13 + TE2(ta1:$).AT21`
- `TE3.AT32` est défini par `DEF= TE2(ta2:$ AND ta1:TE1(:AT11<3)).AT22`

Passons en revue chaque attribut dérivable et analysons les expressions de désignation d'un attribut qui figurent dans leur formule de définition.

Formule de définition de TE1.AT12

On distingue, dans cette formule de définition, deux expressions de désignation d'un attribut.

1. Expression `$.AT13`

On fait appel à `CARD_ATTR` en lui fournissant l'expression à analyser. L'application de l'algorithme sur cette expression peut être représentée comme suit :

```
CARD_ATTR($.AT13)
  CARD_ENSENT($)
    On renvoie [1-1]
  Card=[1-1]
  AT13 est facultatif
  On renvoie [0-1]
```

L'algorithme fournit, pour l'expression `$.AT13`, les cardinalités `[0-1]`. On en conclut que cette expression représente au maximum une valeur et qu'elle est susceptible de n'en représenter aucune (valeur `NULL`).

2. Expression `TE2(ta1:$).AT21`

On fait appel à `CARD_ATTR` en lui fournissant l'expression à analyser. L'application de l'algorithme sur cette expression peut être représentée comme suit :

```
CARD_ATTR(TE2(ta1:$).AT21)
  CARD_ENSENT(TE2(ta1:$))
    TE2(ta1:$) est de la forme TE(Csel)
    CARD_CSEL(ta1:$,TE2)
      ta1:$ est une condition d'association unique.
      CARD_CASS(ta1:$,TE2)
        ta1:$ est de la forme TAssociation:E0
        Card1:=[1-1] (rôle joué par TE1 dans ta1)
        CARD_ENSENT($)
          On renvoie [1-1]
        Card2:=[1-1]
        On renvoie [1-1]
      On renvoie [1-1]
    On renvoie [1-1]
  Card=[1-1]
  AT21 n'est pas facultatif
  On renvoie [1-1]
```

L'algorithme fournit, pour l'expression `TE2(ta1:$).AT21`, les cardinalités `[1-1]`. On en conclut que cette expression représente toujours une et une seule valeur.

Formule de définition de TE3.AT32

On distingue, dans cette formule de définition, une seule expression de désignation d'un attribut.

Expression `TE2(ta2:$ AND ta1:TE1(:AT11<3)).AT22`

On fait appel à `CARD_ATTR` en lui fournissant l'expression à analyser. L'application de l'algorithme sur cette expression peut être représentée comme suit :

CARD_ATTR(*TE2(ta2:$ AND ta1:TE1(:AT11<3)).AT22***)**
- **CARD_ENSENT(***TE2(ta2:$ AND ta1:TE1(:AT11<3))* **)**
  - `TE2(ta2:$ AND ta1:TE1(:AT11<3))` est de la forme `TE(Csel)`
  - **CARD_CSEL(***ta2:$ AND ta1:TE1(:AT11<3), TE2***)**
    - `ta2:$ AND ta1:TE2(:E21<3)` est de la forme $Csel_1$ AND $Csel_2$.
    - **CARD_CSEL(***ta2:$, TE2***)**
      - `ta2:$` est une condition d'association unique.
      - **CARD_CASS(***ta2:$, TE2***)**
        - `ta2:$` est de la forme `TAssociation:`E_0
        - `Card1=[1-1]` (rôle joué par TE3 dans `ta2`)
        - **CARD_ENSENT(***$***)**
          - On renvoie `[1-1]`
        - `Card2=[1-1]`
        - On renvoie `[1-1]`
      - On renvoie `[1-1]`
    - `Card1=[1-1]`
    - **CARD_CSEL(***ta1:TE1(:AT11<3), TE2***)**
      - `ta1:TE1(:AT11<3)` est une condition d'association unique.
      - **CARD_CASS(***ta1:TE1(:AT11<3), TE2***)**
        - `ta1:TE1(:AT11<3)` est de la forme `TAssociation:`E_0
        - `Card1=[1-1]` (rôle joué par `TE1` dans `ta1`)
        - **CARD_ENSENT(***TE1(:AT11<3)***)**
          - `TE1(:E21<3)` est de la forme `TE(Csel)`
          - **CARD_CSEL(***:AT11<3, TE1***)**
            - `:AT11<3` est une condition d'association unique.
            - **CARD_CASS(***:AT11<3, TE1***)**
              - `:AT11<3` est de la forme `:Attribut Rel EnsVal`
              - `AT11` est identifiant de `TE1` mais `Rel` ne vaut pas '='
              - On renvoie `[0-N]`
            - On renvoie `[0-N]`
          - On renvoie `[0-N]`
        - `Card2=[0-N]`
        - On renvoie `[0-N]`
      - On renvoie `[0-N]`
    - `Card2=[0-N]`
    - On renvoie `[0-1]`
  - On renvoie `[0-1]`
- `Card=[0-1]`
- `AT22` n'est pas facultatif
- On renvoie `[0-1]`

L'algorithme fournit pour l'expression `TE2(ta2:$ AND ta1:TE1(:AT11<3)).AT22` les cardinalités `[0-1]`. On en conclut que cette expression représente au maximum une valeur et qu'elle est susceptible de n'en représenter aucune.